Elefant
User's Manual

Release 0.1

Kishor Gawande
Christfried Webers
Alex Smola
Choon Hui Teo
Javen Qinfeng Shi
Julian McAuley
Le Song
Quoc Le
Simon Guenter

September 30, 2009

Statistical Machine Learning Program
National ICT Australia
Locked Bag 8001
Canberra ACT 2601
Australia

CONTENTS

CHAPTER

ONE

Cover Tree

1.1 Overview

A Cover Tree is a datastructure helpful in calculating the nearest neighbor of points given only a metric. A cover tree is particularly motivating for a confluence of reasons:

1. The running time of a nearest neighbor query is only O(log(n)) given a fixed intrinsic dimensionality.
2. The space usage and query time are O(n) under no assumptions. (like the naive approach, and ball trees)
3. It's remarkably fast in practice.

This section of the document outlines the basic instructions for using the coverTree algorithms in Elefant. First is the file hierarchy (section 1.2), which is just a tree of all files and folders used throughout this documentation. Next is the Quick-Start Guide (section 1.3), which describes all of the core commands needed to use the build coverTree, and then speed up KNN algorithm by it. Next (in section 1.4), we look at some more advanced topics, such as how to customize distance function in coverTree, how to read input data from files, how to test time cost. Lastly, we cover some examples (section **??**). The examples should probably be simple enough that they can be read immediately, so feel free to jump right ahead if you're feeling confident (or you're in a hurry).

1.2 File Hierarchy

The files used in this document are all contained in *Elefant/coverTree*. The hierarchy of the files contained therein is as follows:

coverTree:

coverTree.so	The module of coverTree
coverTree.so.1.33.1	The file need to be put int /lib:
coverTree/src:	
coverTree.h	headfile for pyste
coverTree.cpp	the cpp file which was generated by pyste and partially modified manually for boost.bjam
coverTree.pyste	the interface file for pyste
Jamfile	for bjam
Jamerules	bjam rules
coverTree/data:	
**.data*	data sets
coverTree/examples:	
coverTreeTest.py	example for using coverTree to speed up knn
coverTree/doc:	
**.tex*	documentation

1.3 Quick-Start Guide

This subsection shows how to install, then import and then use coverTree.

1.3.1 Installation

1. If you have coverTree.so (or coverTree.dll) file, just copy libboost_python.so.1.33.1 into your direcotry (such as /lib:) and add *export LD_LIBRARY_PATH=your directory*into bash and run bash again.

2. If you don't have coverTree.so (or coverTree.dll)file, firstly you have to install boost. It also means you need set some envirnment varibles for it. For instance if your boost is installed in */boost_1_33_1/* and your python is version 2.3, you need *export BOOST_BUILD_PATH= /boost_1_33_1/,export PYTHON_VERSION=2.3*. This work is typical step for boost.python, you can find how to use it if you still have problem. After you install boost successfully, you can compile coverTree from source */elefant/coverTree/src*, by running command *bjam -t coverTree.cpp*. Then it will produce coverTree.so (or coverTree.dll) file.

1.3.2 Use coverTree

Because coverTree module support input data from both file and numpy.array. So if you want to use numpy.array type, you need to import both coverTree and numpy.

```
import numpy as N
import elefant.coverTree.coverTree as C
```

The simplest way to use build a coverTree is build it from the existing numpy.array in python.

```
theArray=N.arange(40,dtype='Float32').reshape(10,4)
a = C.CCoverTree(theArray)
```

This is how to build coverTree using the default metric(Norm2 distance function). Users can customize their own defined metric as well which will be introduced in 1.4. The way to search the k nearest points of set b in the set a, is buid another coverTree b, then a.knn(b,k).

```
theArray2=N.arange(12,dtype='Float32').reshape(3,4)
b = C.CCoverTree(theArray2)
a.KNearestNeighbor(b,1)
```

The query result is:

```
Printing results
8.000000 9.000000 10.000000 11.000000 0.000000 0.000000 0.000000 0.000000
8.000000 9.000000 10.000000 11.000000 0.000000 0.000000 0.000000 0.000000

4.000000 5.000000 6.000000 7.000000 0.000000 0.000000 0.000000 0.000000
4.000000 5.000000 6.000000 7.000000 0.000000 0.000000 0.000000 0.000000

0.000000 1.000000 2.000000 3.000000 0.000000 0.000000 0.000000 0.000000
0.000000 1.000000 2.000000 3.000000 0.000000 0.000000 0.000000 0.000000

results printed
```

The first point (represented as vector) is the object point in b. The following point(if $k = 1$) or points(if $k > 1$) is the corresponding k nearest points in a. The return result is a list. Every element of the list is a numpy.array.

```
[array([[  8.,   9.,  10.,  11.,   0.,   0.,   0.,   0.],
       [  8.,   9.,  10.,  11.,   0.,   0.,   0.,   0.]], dtype=float32),
 array([[ 4.,  5.,  6.,  7.,  0.,  0.,  0.,  0.],
       [ 4.,  5.,  6.,  7.,  0.,  0.,  0.,  0.]], dtype=float32),
 array([[ 0.,  1.,  2.,  3.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  2.,  3.,  0.,  0.,  0.,  0.]], dtype=float32)]
```

1.4 Advanced Topics

This section we will introduce how to customize your distance function, how to get speedup factor, how to build coverTree in different ways.

1.4.1 Customize your distance function

You can customize your distance function and use it when you search by coverTree. Essentially, every function object you can load in python can be pass to coverTree constructor as your distance. So you can define he distance function in python, or in c extension. Please notice that our core part is written by c++ for speed reason, so if you define your distance in python, it will be slower than the one in c extension and much slower than our default distance function(which has bee optimized in c++). You can see the sorce code and use the same way with default distance function to gain the fastest speed. The simplest way is that define your distance function in python, just like

```
# customized distance function 1
def MyDist(v1, v2, upBound):
  length = len(v1);
  sum = 0;
  upBound2 = upBound*upBound;
  for i in range(length):
    a = v1[i] -v2[i];
    a = a*a;
    sum = sum+a;
    if sum>upBound2:
      return upBound+1;
  return math.sqrt(sum)
```

Note that you have to add a small number at the end of upBound such as 1, otherwise it will cause a unexpected result. This is caused by Cover Tree C++ implementation. After you defined your distance, you just need pass it as a parameter to the coverTree constructor.

```
a = C.CCoverTree(theArray,mydist)
```

The second way is to define it in a c extension. You may like to see how to write c extension firstly. Then use *bjam -t distLib.cpp* to build it into a single *.so file. More details see developer manul.

1.4.2 Get speedup factor

Using *TestTime()* to see how much coverTree can speed up KNN in your machine. Following codes show how to call *TestTime()* in different ways.

```
#test coverTree based knn and brute knn computation time and factor of speed up.
C.TestTime(3,theArray);
C.TestTime(3,theArray,C.Norm2StdVec);
C.TestTime(3,theArray,MyDist);
C.TestTime(3,'../data/test.data')
C.TestTime(3,'../data/test.data',C.Norm2StdVec)
C.TestTime(3,'../data/test.data',MyDist)
```

The result is shown like

```
Size    CoverTree time  Brute time      Speed up
20000   4.793043        71.785217       14.976960
```

The larger data set is, the more it can be speed up.

1.4.3 Different ways to build coverTree

Build coverTree from numpy.ndarray with different distance functions.

```
theArray=N.arange(40,dtype='Float32').reshape(10,4)
a = C.CCoverTree(theArray)
b = C.CCoverTree(theArray)
a = C.CCoverTree(theArray,C.Norm1StdVec)
b = C.CCoverTree(theArray,C.Norm1StdVec)
a = C.CCoverTree(theArray,C.Norm2StdVec)
b = C.CCoverTree(theArray,C.Norm2StdVec)
a = C.CCoverTree(theArray,MyDist)
b = C.CCoverTree(theArray,MyDist)
a.KNearestNeighbor(b,1)
```

Build coverTree from data file with different distance functions.

```
a = C.CCoverTree('../data/test.data')
b = C.CCoverTree('../data/test.data')
r1 = a.KNearestNeighbor(b,2)

a = C.CCoverTree('../data/test.data',C.Norm2StdVec)
b = C.CCoverTree('../data/test.data',C.Norm2StdVec)
r2 = a.KNearestNeighbor(b,2)

a = C.CCoverTree('../data/test.data',MyDist)
b = C.CCoverTree('../data/test.data',MyDist)

r3 = a.KNearestNeighbor(b,2)
```

More sophiscated building coverTree step by step.

```
# creat a instance
a = C.CCoverTree()

#:read the data set which can be searched from
pointSet = C.ParsePoints("../data/wine.data")
#read the target data you want to query from wine.data
pointSetQuery = C.ParsePoints("../data/wine_less.data")

#print
pointSetQuery.PrintAll();

#create coverTree for ponitSet, and return the top node.
top = a.BatchCreate(pointSet)

#create coverTree for pointSetQuery, and return the top node.
topQuery = a.BatchCreate(pointSetQuery)

# k parameter for KNN
k = 3

#customized type to store result.
result = C.VArrayResult()

#result is returned to result varialbe
a.KNearestNeighbor(top,topQuery,result,k)

#print result
C.PrintResult(result)
```

1.5 Examples

The following is a simple example to use coverTree.

```
import numpy as N
import elefant.coverTree.coverTree as C
theArray=N.arange(40,dtype='Float32').reshape(10,4)

a = C.CCoverTree(theArray)
b = C.CCoverTree(theArray)

a.KNearestNeighbor(b,1)

C.TestTime(1,theArray)
```

More detail can be found in *coverTree/examples/coverTreeTest.py*. Data set can be found in *coverTree/data/*.

CHAPTER

TWO

Belief Propagation

2.1 User-Level Documentation

2.1.1 Overview

This section of the document outlines the basic instructions for using the propagation algorithms in Elefant. First is the file hierarchy (section 2.1.2), which is just a tree of all files and folders used throughout this documentation. Next is the Quick-Start Guide (section 2.1.3), which describes all of the core classes needed to use the junction-tree algorithm, and loopy belief-propagation. Next (in section 2.1.4), we look at some more advanced topics, such as sparse representations, and marginalisation. Next (in section 2.1.6), we look at some of the other algorithms, such as 'From Fields to Trees', and EM for HMMs. Lastly, we cover some examples (section 2.1.7). The examples should probably be simple enough that they can be read immediately, so feel free to jump right ahead if you're feeling confident (or you're in a hurry).

2.1.2 File Hierarchy

The files used in this document are all contained in *beliefprop/core*, in the *crf* directory of Elefant. The hierarchy of the files contained therein is as follows:

core:

em.py	The Expectation-Maximisation algorithm (section 2.1.6)
fieldstotrees.py	The Fields to Trees algorithm (section 2.1.6)
junctiontree.py	The Junction-Tree algorithm (section 2.1.3)
loopybp.py	The Loopy Belief-Propagation algorithm (section 2.1.3)
loopybp_pypar.py	Concurrent Loopy Belief-Propagation (section 2.1.4
factorgraph.py	Factor-Graphs (section 2.1.6)
core/errors:	
propagationerrors.py	Errors and warnings (not documented here)
core/models:	
discrete.py	The discrete model (used throughout)
discrete_numpy.py	The discrete model, implemented using numpy (section 2.1.5)
discrete_sparse.py	The sparse model (section 2.1.5)
gaussian.py	Nonparametric BP, using Gaussians (section 2.1.5)
interface.py	The interface used by these models (section 2.2.3)
semirings.py	The semirings used by these models (section 2.2.2)
core/structures:	
cliques.py	Cliques, and their connections (section 2.1.3)
cliques_pypar.py	Cliques, concurrent version (section 2.1.4)
toplevel.py	Parent classes for all algorithms (not documented here)
utilities.py	Some basic utilities (not documented here)

These modules can now be imported using the paths given. When importing models, it is better to use *"import modelname"* rather than *"from modelname import *"*, since each model implements similar functions, and there may be name-clashes otherwise. In all of the following examples, it is assumed that the appropriate import statements have already been executed.

2.1.3 Quick-Start Guide

This guide should be sufficient in order to begin using the algorithms as quickly as possible. We'll begin by just looking at the classes that the user needs, and some simple code segments. Most of the classes used in the following pages are defined in *core.models.discrete*. Only the junction-tree algorithm, and loopy belief-propagation are considered here. Other algorithms (such as 'From Fields to Trees', and EM for HMMs) are covered in section 2.1.6.

If you're feeling confident, you might skip this section altogether, and proceed straight to some of the simpler examples – these are covered in section 2.1.7.

Nodes

A node is nothing more than a container for a domain. For a discrete-valued problem, a domain is nothing more than a list. Hence, all that is needed to initialise a node is a domain.

For example, the following code could be used to create some nodes:

```
n1 = discrete.Node([1,2,3])
n2 = discrete.Node(["woebegone", "forlorn"])
n3 = discrete.Node([0])
```

Potential Functions

Before we can define a clique, we need to define a potential function that will operate on its nodes, and a semiring that will operate on the resulting distribution. Most of the standard semirings are already implemented in *core.models.semirings*, but we still need to define our own potential functions.

A potential function will act on a set of nodes. Each of the arguments to a potential function should therefore be an element of one of its nodes' domains. The potential function should simply return a potential, which may be a number, or a truth value, etc. – whatever works with our chosen semiring.

For example, if we have defined some nodes:

```
n1 = discrete.Node(["happy", "sad"])
n2 = discrete.Node(["wet", "dry"])
```

then a two-dimensional potential function corresponding to these nodes may be:

```
def pfunction(x, y):
  return {("happy", "wet"):0.2, ("happy", "dry"):0.4,
          ("sad", "wet"):0.25, ("sad", "dry"):0.15}[(x,y)]
```

Or, we might have:

```
n1 = discrete.Node(range(20))
n2 = discrete.Node(range(20))

def pfunction(x, y):
  return x + x*y + 1
```

Now, to make this potential function usable by a clique, we have to create an instance of the *Potentials* class (again in *core.models.discrete*), along with the semiring we want to work with. This will ensure that each clique is able to calculate its distribution when it needs to.

For example:

```
pot = discrete.Potentials(pfunction, semirings.semiring_sumproduct)
```

Cliques

Having defined some potential functions and some nodes as above, we are now able to build some cliques. For each clique, we need only to know which potential function it is using, and the nodes that it contains (note that the nodes will be passed to the potential function in the same order as they are passed to the initialiser).

For instance, suppose we have defined the following nodes and potential functions:

```
n1 = discrete.Node(range(5))
n2 = discrete.Node(range(10))
n3 = discrete.Node(range(15))
n4 = discrete.Node(range(20))
n5 = discrete.Node(range(25))

def pfunction1(x, y, z):
  return x*y + z + 1

def pfunction2(x, y):
  return exp(-(x-y)*(x-y))

pot1 = discrete.Potentials(pfunction1, semirings.semiring_sumproduct)
pot2 = discrete.Potentials(pfunction2, semirings.semiring_sumproduct)
```

we can then define some cliques simply by passing a set of nodes and a potential function to the initialiser for a clique (defined in *core.structures.cliques*):

```
c1 = cliques.Clique([n1,n2,n3], pot1)
c2 = cliques.Clique([n3,n4], pot2)
c3 = cliques.Clique([n3,n4], pot2)
```

Estimators

Before we can run a belief-propagation algorithm, we need first to define how estimates are going to be made from marginal distributions. Some simple estimators, such as a sampler, and maximum a posteriori (MAP) estimation, are defined in *core.models.discrete*. These are likely to work with most distribution types, so we will defer further explanation until later. For now, it should suffice to know that an 'estimator' is simply a method which takes a distribution, and returns some assignment to its nodes.

Junction-Tree Algorithm

Now, creating a junction-tree algorithm is simply a matter of passing our cliques, and our estimator to the *JunctionTreeAlgorithm* class (defined in *core.junctiontree*). For example:

```
c1 = cliques.Clique([n1,n2,n3], pot1)
c2 = cliques.Clique([n3,n4], pot2)
c3 = cliques.Clique([n3,n4], pot2)

jt = junctiontree.JunctionTreeAlgorithm([c1,c2,c3], discrete.MAPestimate)
```

To send messages throughout the tree, simply call *JunctionTreeAlgorithm.propagate*, for example:

```
jt.propagate(progress = True)
```

This will propagate all messages throughout the tree, printing progress as we go (progress is not printed by default).

Loopy Belief-Propagation

Using the loopy belief-propagation algorithm is no different from using the junction-tree algorithm, except that we now propagate for a certain number of iterations.

```
lbp = loopybp.LoopyAlgorithm([c1,c2,c3,c4], discrete.MAPestimate)
lbp.propagate(iterations = 10)
```

Extracting Results

Having propagated all messages throughout the tree, we may now take estimates from the marginal distributions for any of its nodes. We simply pass a list of nodes to *JunctionTreeAlgorithm.findresults*, and it will return a dictionary, mapping nodes to their corresponding estimates. The estimates are made using whatever estimator was passed to the constructor for the clique containing that node.[1] Of course, every estimate is simply an element of that node's domain.

```
results = jt.findresults([n1,n2,n3])

print results[n1], results[n2], results[n3]
```

[1] Actually, this method does not allow us to choose *which* clique's distribution is used to make the estimate, but this should not be an issue for the junction-tree algorithm. It is possible to use a clique of our choice – this will be covered in Advanced Topics (section 2.1.4).

2.1.4 Advanced Topics

Working in the Log-Domain

Repeatedly marginalising and multiplying distributions together has the risk of resulting in overflow or underflow. In principle, this is not a concern, as it we may just restrict ourselves to dealing with 'normalised' distributions. However, normalisation is a very costly procedure, meaning that a great deal of computational resources can be saved if we can avoid it.

As a result, it may be desirable to perform all of our semiring operations in the log-domain, meaning that we can avoid overflow, without having to normalise our distributions.

Therefore, where possible, the standard semirings have been re-implemented to work in the log-domain[2] – just import *semirings.semiring_logsumproduct* instead of *semirings.semiring_sumproduct* etc. Normalisation will *not* be performed with these semirings (see section 2.2.2 for further explanation).

Additionally, it is necessary to specify different estimators to be used in the log-domain. Instead of using *discrete.sample*, *discrete.logsample* must be used instead (likewise for other estimators). This does nothing more than exponentiate the distribution before making the estimate.

Marginals in Multiple Dimensions

Since we may sometimes want the marginal with respect to many nodes, rather than only a single node, the method *findmultimarginal* (available for any of the basic algorithms) has been provided. Of course, it is only possible to find marginals of several nodes if they are all contained within a single clique. If there is no clique containing these nodes, then an error will be raised.

Of course, the value returned is a distribution, of whatever type has been selected by the user. Therefore, this method should only be used with some caution, since it exposes the internal representation of a distribution to the caller. It's probably more meaningful to pass the result to an estimation function.

For example, suppose we have a clique containing the nodes *n1, n2* and *n3* (somewhere in *jt*, our junction-tree). Then we might call:

```
marginal = jt.findmultimarginal([n2, n1])
est = discrete.MAPestimate(marginal)
```

Now, *est* will contain some assignment to *n2* and *n1*, in that order.

Marginalisation

Sometimes, we may wish to further marginalise a distribution, for example one found using the *findmultimarginal* function described above (section 2.1.4). This is simple, since any *Distribution* class must implement a *marginalise* function. This method may simply be passed to the nodes whose marginals we want. For example, suppose we find a multi-dimensional marginal distribution, using:

```
dist = jt.findmultimarginal([n1, n2, n3])
```

Then we can further marginalise this distribution by calling:

[2] However, it is worth mentioning that there are some cases where this is not possible; for example, when using the *numpy* semirings (section 2.1.5), there is no efficient way to compute a sum in the log-domain.

```
dist.marginalise([n3, n1])
```

Note that the *marginalise* function provided by the discrete model is able to change the node-ordering in the above two marginals.[3]

Passing Messages Efficiently

Rather than simply calling *junctiontree.propagate*, which passes *all* messages, in both directions throughout a tree, it may be desirable to pass messages only in one direction, if we only require the marginal for a single node (or a whole clique). This can be done using *JunctionTreeAlgorithm.onemarginaldistribution*, along with the node whose marginal we want. As with the previous example (2.1.4), this exposes the internal representation of the distribution to the caller, so the result should probably be passed to an estimation function. For example:

```
marginal = jt.onemarginaldistribution(n1)
est = discrete.sample(marginal)
```

This method will simply search for any clique containing the desired node, and pass messages to that clique. Since this may be suboptimal in many cases, we are also able to explicitly specify a clique to which messages must be passed (of course, the selected clique must contain the selected node). For example:

```
marginal = jt.onemarginaldistribution(n1, clique_containing_n1)
```

Of course, if we call this method on several different cliques, each subsequent call will make use of the messages which have already been passed.

Changing a Node's Domain

The domain being used by a node may be changed by a user at any time. This method is typically only accessed by the algorithms themselves, and not by the user, but there may nevertheless be call to use it (or a use to call it).

A node's distribution is changed by calling *Node.setdomain*.[4] For example:

```
n1.setdomain(range(10))
n1.setdomain([2])
```

Of course, doing this invalidates the distributions stored for any cliques containing that node, and indeed any marginal distributions that clique has passed as messages. While an individual clique may have its distribution purged by calling *Clique.clear*, it is probably more prudent to clear all distributions and messages in the entire tree, as they will all have been invalidated by this procedure. This is done by calling *JunctionTreeAlgorithm.clearall*, or *LoopyAlgorithm.clearall*. For example:

[3] Of course, when it comes time to implement *your own* model, this generality need not be maintained.
[4] Assuming it is implemented by the model of a node being used – see section 2.2.3

```
lbp.propagate(10) # Propagate for 10 iterations.
n1.setdomain([5])
lbp.clearall()
lbp.propagate(10)
```

Actually, if we are setting a node's domain to be only a single value, this is the same as saying that this is an 'observed' node. This can actually be done using the methods *Node.fixvalue* and *Node.unfixvalue*, assuming that they are implemented by the model of a node being used. It is also worth mentioning, that in addition to the *clearall* method, there is also a *clear* method – this can be used to clear the messages, without destroying the local distributions for each clique. While this method is not appropriate here (changing a node's domain invalidates its clique's local distribution), it may be desirable in some cases.

Alternate Stopping Conditions

By default, the Loopy Belief-Propagation algorithm simply runs for a given number of iterations, as passed to its *propagate* method. In some cases, we may instead want to cease propagation once all message differences are below a certain threshold.

For this reason, the *Distribution* class (as defined in *discrete.Distribution* etc.) implements a method to determine the difference between messages. The default difference measure is simply the sum of the differences for each value in the distribution's domain. If a different measure is desired, it will have to be implemented by the user – this is described in section 2.2.3.

Assuming that this difference measure is correct, it is simply a matter of passing a 'cutoff' parameter to *LoopyAlgorithm.propagate*, specifying the minimum message difference required for a message update to occur. Once all message differences are below this cutoff value, propagation will cease. For example:

```
lbp.propagate(cutoff = 0.1)
```

will simply continue propagation until all message differences are below 0.1. Alternately:

```
lbp.propagate(iterations = 20, cutoff = 0.1)
```

will do the same, but will only run for a maximum of 20 iterations, even if some message differences are above this threshold.

Dealing with Node/Edge Features

It may often be the case that the potential functions in our field are homogenous, yet differ based on the values of certain node or edge 'features'. A simple way to deal with these features would simply to be to extend each of our cliques to include a 'feature-node', which would have a fixed value, containing the node and edge features for that clique. Another option would be to define a different potential function for each clique, which takes into account the different values of these features. While both of these are perfectly correct solutions, it may occasionally be somewhat cumbersome to implement them.

Therefore, another option is included, which allows us to pass the node objects directly to the potential function. This allows us to retrieve the exact nodes the potential function is being called on, in the event that this potential function is being used for many cliques simultaneously.

To facilitate this, the potential function in the interface *interface.BasicPotentials*, and all derivative potential classes, include an optional argument, `passnodes`. By setting `passnodes = True`, the node objects will be passed as

extra arguments to our potential functions. Thus, where we had previously used

```
def pot(x,y,z):
  return x*y + z # for example.

p = discrete.Potentials(pot, semirings.semiring_sumproduct)
```

we would now use something more like

```
nodefeatures = {node1: 4, node2: 5, node3: 6} # for example

def pot(x,y,z,nx,ny,nz):
  return x*y + z + 2*nodefeatures[nx]

p = discrete.Potentials(pot, semirings.semiring_sumproduct, passnodes = True)
```

Concurrent Loopy Belief-Propagation

An extension of the loopy belief-propagation algorithm (*core.loopybp*) allows it to be distributed among several processors/machines. The current implementation uses *pypar*, a Python binding for MPI available at

`http://datamining.anu.edu.au/~ole/pypar/`

Using this algorithm will also require that MPI is installed (this implementation has been tested using *mpich2*), and has been activated on several machines.

To use this algorithm, *pypar* must first be imported:

```
import pypar
```

Also, when importing *cliques* and *loopybp*, the *pypar* versions must instead be imported:

```
from elefant.crf.beliefprop.core import loopybp_pypar as loopybp
from elefant.crf.beliefprop.core.structures import cliques_pypar as cliques
```

Furthermore, when the script is finished, *pypar.finalize* must be called:

```
pypar.finalize()
```

Now, the main difference when using this algorithm is that each clique must be assigned to a processor (the total number of processors can be determined using *pypar.rank*). For instance, assigning a clique to the 1st processor would become:

```
c1 = cliques.Clique(0, # Processor number.
                    [node1, node2],
                    potential_function)
```

After this has been done for all cliques in the model, an instance of *loopybp_pypar.LoopyAlgorithm* may be created. Initialisation of this algorithm is done in exactly the same way as *loopybp.LoopyAlgorithm*:

```
lpb = loopybp.LoopyAlgorithm([c1, c2, c2], discrete.MAPestimate)
```

However, this initialiser does include two additional keyword arguments. Firstly, *rootcpu* allows us to select which cpu

will be used to perform certain tasks that cannot be distributed, such as determining the message passing order, and collecting the results at the end. By default, cpu '0' is used, but a different one may be specified. As this constitutes only a small part of the algorithm, it should make little difference which cpu is chosen.

Secondly, when one processor is scheduled to receive a message from another processor, we have the option of either waiting for that processor to send the message (and blocking until then), or simply continuing without this message. In the latter case, an additional thread is spawned to wait for the message's eventual arrival. While the latter case is often significantly faster, it may also be slightly less accurate. This option is specified by setting *allowmissing* to *True* or *False* (default is *True*). For instance, another possible initialisation might be:

```
lpb = loopybp.LoopyAlgorithm([c1, c2, c3],
                             discrete.MAPestimate,
                             allowmissing = False,
                             rootcpu = 3)
```

The final difference is that once the results are collected after propagation, they are only collected by a single processor. Calling

```
results = lpb.findresults([n1, n2, n3])
```

will cause only one cpu (the 'root' cpu, specified above) to receive a full set of results, while all others simply have *None* returned. Printing the results might then be done by:

```
results = lpb.findresults([n1, n2, n3])
if (pypar.rank() == 0): # If we are the root cpu
  print results[n1] # etc...
```

Finally, a word of caution should be issued when using these algorithms – since belief-propagation often involves large messages being passed (e.g. multidimensional matrices), there is a significant cost associated when passing messages between machines using MPI. This implementation should therefore only be considered when the size of the messages is reasonably small.

2.1.5 Different Distribution Models

The Sparse Model

In many cases, it may be desirable to reject certain low-probability elements from a distribution, in order to minimise the size of the messages being passed. To this end, the existing discrete model has been extended to handle sparse representations.

By default, the sparse model can be used in exactly the same way as the existing discrete model, and it will reject any configurations with probability 0 (or False). If this is the desired behaviour, all that is required is a different 'import' statement (i.e. import *discrete_sparse* rather than *discrete*).

If this is not the desired behaviour, it is possible for users to specify explicitly which configurations are to be rejected. All that is needed is a function that takes a probability (or more generally, an element of our semiring), and returns 'False' if this probability is too low (and 'True' otherwise).

This function is now passed as the *last* argument to the initialiser for *discrete_sparse.Potentials* (see 'Potential Functions', section 2.1.3).

For example, suppose we want to accept only those configurations with probability greater than 0.01. Then our rejection function would be:

```
def rejection_function(x):
  return (x > 0.01)
```

Suppose also that we have defined some potential function for our clique, such as:

```
def pfunction(x, y):
  return math.exp(-((x-y)**2))
```

Now, our call to *discrete_sparse.Potentials* may become:

```
pot1 = discrete_sparse.Potentials(pfunction,
                                  semirings.semiring_sumproduct,
                                  rejection_function)
```

The sparse model will always ensure that this rejection-function is only called on *normalised* distributions (as long as normalisation is possible, see section 2.2.2 for more details). Indeed, it would likely be otherwise impossible to define such a function if the magnitude of the distributions was unconstrained.

The Numpy Model

In the interest of performance, the discrete model has also been re-implemented to use *numpy* (a Python array library). In most cases, using this model should be identical to using the discrete model, with only two minor differences – firstly, we obviously require a different import statement (i.e. we must import *discrete_numpy*, rather than *discrete*); secondly, we cannot use the same semirings as we did for the standard discrete model.

Fortunately, all of the standard semirings have also been re-implemented to use *numpy*. Simply use *semirings.semiring_numpy_sumproduct* as opposed to *semirings.semiring_sumproduct*, etc.

The Gaussian Model

In order to perform nonparametric belief-propagation, a model is included in which each potential function takes the form of a Gaussian mixture. This type of model is implemented in *gaussian* (in *core.models*).

Firstly, to create each Gaussian in our mixture, we call *gaussian.Gaussian* with a mean and covariance matrix (or its inverse). Since we have a *mixture* of Gaussians, each Gaussian also has a relative importance. An example of such a Gaussian would be

```
g = gaussian.Gaussian(0.1,                          # Importance
                      numpy.array([1,2,3]),         # Mean
                      cov = numpy.array([[1,2,3],
                                         [2,3,4],
                                         [3,4,5]])) # Covariance
```

A Gaussian can be specified using either its covariance matrix (by setting `cov = ...`) or using the *inverse* of its covariance matrix (by setting `covinv = ...`). At least one of these two keyword arguments must be used. The importance terms will be appropriately scaled once our mixture is created (see below).

Creating a potential function is very similar to the discrete model, except that this time, we pass a collection of Gaussians in the place of a function object; we also ignore the semiring argument. Additionally, to prevent the number of Gaussians in our mixture becoming too large (as would happen with repeated multiplications), we specify the maximum number of Gaussians to be allowed in our mixture. Now, upon multiplication, only the *most important* Gaussians will be maintained.

An example potential function is created as follows:

```
prior = gaussian.Potentials([gaussian1, gaussian2, gaussian3],
                            maxgaussians = 9)
```

Another issue when dealing with Gaussian distributions is that of estimation after propagation. Sampling, or mode-finding in Gaussian mixtures is actually quite difficult. For this reason, a utility function, *gaussian.Distribution.maxestimate1d*, has been included, which simply returns the highest probability assignment within a given, discrete, domain. For example, once we have determined the final distributions (using *findresults*, for example – see section 2.1.3), we might call

```
max = results[n].maxestimate1d(range(256))
```

which would choose the highest probability assignment out of the integers from 0 to 255.

2.1.6 Other Algorithms

From Fields to Trees

'From Fields to Trees' refers to an algorithm devised by Hamze and Freitas, in which a field containing loops is decomposed into several trees. Inference is now done using the junction-tree algorithm, on one tree at a time, using the current estimates for any adjoining trees as belief nodes.

In order to use this algorithm, we need to decide upon some decomposition of our field into a set of trees. There may be any number of trees, and they may be overlapping – the only requirement is that they each satisfy the junction-tree property (otherwise an error will be raised).

Each tree is just a set of cliques. For example, suppose we have a simple 'loop', such as:

```
c1 = cliques.Clique([n1, n2], pot1)
c2 = cliques.Clique([n2, n3], pot1)
c3 = cliques.Clique([n3, n4], pot1)
c4 = cliques.Clique([n4, n1], pot1)
```

Then one possible decomposition would be:

```
subtrees = [[n1, n2], [n3, n4]]
```

Here, the tree containing *n1* and *n2* will see *n3* and *n4* as belief nodes, and vice versa.

Hamze and Freitas proposed that the inferred values should be found by taking a Rao-Blackwellised estimate, which in this case is the expected-value of the sum of the estimates for each iteration. Of course, unless we are using a real-valued model, it is not necessarily clear exactly what is meant by 'expected-value'. Therefore, the user may have to specify their own expected value function. The provided function (*discrete.expectedvalue*) only works on discrete, real-valued distributions, and may indeed return an estimate that is *not* in the discrete domain. Whether this behaviour is desirable or not is up to the application at hand.

Otherwise, the initialiser (for *fieldstotrees.FieldsToTreesAlgorithm*), takes a list of *all* cliques, a list of subtrees (as above), an expected-value function (such as *discrete.expectedvalue*), and an estimator (such as *discrete.sample*). For example, using the above cliques and subtrees, our algorithm would be created using:

```
ftta = fieldstotrees.FieldsToTreesAlgorithm([c1, c2, c3, c4],
                                            subtrees,
                                            discrete.expectedvalue,
                                            discrete.sample)
```

As with loopy belief-propagation, inference is performed by using *FieldsToTreesAlgorithm.propagate*, with a chosen number of iterations.[5] Similarly, the *findresults* method is implemented for this class, so inference here is exactly the same as it is in section 2.1.3.

Additionally, we may wish to use some 'initial estimates', for the starting values of our belief nodes. This is done by passing a dictionary of (*Node*:*value*) pairs to the initialiser (as the last argument). Any nodes without an estimate will simply be initialised to any random value in their domain.

For example, the above line of code may become:

[5] Here, an 'iteration' means performing the junction-tree algorithm on *every* tree in the field, exactly once, in the order that they were passed to the initialiser.

```
ftta = algorithms.FieldsToTreesAlgorithm([c1, c2, c3, c4],
                                         subtrees,
                                         discrete.expectedvalue,
                                         discrete.sample,
                                         {n1:4, n2:5, n4:3})
```

EM for HMMs

In order to learn potential functions, we may want to use the Expectation-Maximisation (EM) algorithm, for both decomposable and non-decomposable Hidden-Markov-Models (HMMs).

As usual, we begin by defining the topology of our graph. The simplest *non*-decomposable case is just a loop containing four nodes:[6]

```
n1 = discrete.Node(["wet", "dry"])
n2 = discrete.Node(["bicycle", "bus"])
n3 = discrete.Node(["early", "late"])
n4 = discrete.Node(["happy", "sad"])

c1 = cliques.Clique([n1, n2], None)
c2 = cliques.Clique([n2, n3], None)
c3 = cliques.Clique([n3, n4], None)
c4 = cliques.Clique([n4, n1], None)
```

Note that instead of passing a potential function to the cliques, we simply pass 'None'. We do this because the potential functions are initially unknown. Alternately, if we had some initial estimates for our potential functions, it would be perfectly acceptable to initialise using these estimates instead of 'None'.

Next, we need a list of observations. Of course, we are able to deal with the case in which some of the observations are not complete (or 'fully observed'). Each of our observations should just be a dictionary mapping nodes to observed values (values in their domains). For example, a set of observations may be:

```
observations = [{n1:"wet", n2:"bicycle", n3:"late", n4:"sad"},
                {n2:"bicycle", n3:"early", n4:"happy"},
                {n1:"wet", n2:"bicycle", n3:"early", n4:"sad"},
                # ....
                {n1:"wet", n2:"bus", n4:"sad"},
                {n1:"wet", n3:"late", n4:"sad"}]
```

Now, to the initialiser (of *em.EMNondecomposable*), we must pass the model we are using, the full set of cliques, our set of observations, the number of iterations to be used by the iterative-proportional-fitting (IPF) procedure, the number of iterations to be used by loopy belief-propagation, the semiring we want to use, and finally a 'flat' function. The 'flat' function should simply return something analogous to the '1' element of our chosen semiring, so that we can make a uniform distribution. This will be used as an initialisation for any potential functions for which we have provided no initial estimates. Naturally, for the sum or max-product semirings, this function just returns 1. The default flat function just returns 1.

Hence, a valid initialisation may be:

[6] Most of this code is taken from *example3.py*, which deals with this algorithm.

```
# Other initialisation code...

ema = em.EMNondecomposable(discrete,
                           observations,
                           10, # IPF updates
                           10, # Iterations of LBP
                           discrete.MAPestimate,
                           semirings.semiring_maxproduct)
```

Now, we can just call *EMNondecomposable.iterate* to perform learning, for the desired number of iterations:

```
ema.iterate(10)
```

Having performed this procedure, the cliques will now have the learned potential functions. These can be extracted explicitly if they are desired (using *Clique.getpcalculator*), or we can just use the cliques directly in our inference procedures. For example, if we tried:

```
lbp = algorithms.LoopyAlgorithm([c1, c2, c3, c4], discrete.MAPestimate)
```

This would then use the updated potential functions, even though they were initialised with 'None'.

Factor Graphs

While factor graphs differ significantly from other belief-propagation algorithms, from the user's perspective they are almost identical to the loopy belief-propagation and junction-tree algorithms. Instead of a clique, we now have a *Factor* (*factorgraph.Factor*), which still takes a set of nodes and a potential-function as its initialiser:

```
f = factorgraph.Factor([node1,node2,node3], pot)
```

To create a factor graph, we initialise an instance of *factorgraph.LoopyFG* or *factorgraph.JunctionTreeFG* with our set of factors, and an estimator (much as we did with *loopybp.LoopyAlgorithm* nad *junctiontree.JunctionTreeAlgorithm*):

```
alg = factorgraph.LoopyFG([factor1, factor2, factor3], discrete.MAPestimate)
```

The interface presented to the user is otherwise identical to that for *loopybp.LoopyAlgorithm* or *junctiontree.JunctionTreeAlgorithm*.

2.1.7 Examples

Here, we will guide the reader through some simple example programs. Substantially more complete examples are provided along with the code itself, but these are rather large, and too complicated to include here. The code used in this example is included in the *examples* directory.

Junction-Tree Algorithm

Suppose we wish to optimise the parameters for a date (i.e. maximise the likelihood of further dates occurring). We want to see the best movie, wear the best clothes, and go to the best restaurant, yet simultaneously minimise the cost to ourselves.

Suppose we have determined the following relationships between our variables: The price, our clothes, and the movie we see are all dependent on our choice of restaurant, but these three variables are *conditionally independent*, given our choice of restaurant. Hence the topology of our graphical model is as follows:

```
#              price
#                |
# clothes----restaurant----movie
```

First, we need to import the required modules:

```
from elefant.crf.beliefprop.core import junctiontree
from elefant.crf.beliefprop.core.models import discrete
from elefant.crf.beliefprop.core.models import semirings
from elefant.crf.beliefprop.core.structures import cliques
```

Next, we define the prices, clothes, restaurants, and movies that we may choose:

```
prices = range(100)

clothes = ["shorts and t-shirt",
           "dinner suit",
           "tuxedo",
           "tracksuit"]

restaurants = ["McDonalds",
               "L'Escargot",
               "Pete's Pork Palace"]

movies = ["Babe",
          "Supersize Me",
          "Star Wars Episode II",
          "Lord of the Rings"]
```

Now, each of these variables defines a *node* in our model. Hence we create four nodes:

```
node_price = discrete.Node(prices)
node_clothes = discrete.Node(clothes)
node_restaurant = discrete.Node(restaurants)
node_movie = discrete.Node(movies)
```

Next, we must define the potential functions that operate on the cliques (as defined by the above topology). These potential functions simply assign a non-negative potential to each possible combination of values. First is the potential acting between prices and restaurants:

```
def price_restaurant(p, r):
  if (r == "McDonalds"):
    minprice = 5
    maxprice = 20
  elif (r == "L'Escargot"):
    minprice = 70
    maxprice = 100
  elif (r == "Pete's Pork Palace"):
    minprice = 20
    maxprice = 40

  if (p < minprice):
    return 0
  elif (p > maxprice):
    return 0
  else:
    return 2500 - (p-50)**2
```

Here, we assign a potential of 0 if the price is not within the price-range for the chosen restaurant. Otherwise, the potential is determined by a quadratic function, which tries to balance the fact that our date will not be impressed if we spend a small amount of money, and that we will not be impressed if we spend too much money. Ultimately, 0 or 100 dollars each have a potential of zero, and all other amounts are assigned some non-negative value.[7]

Next is the potential between clothes and restaurants:

```
def clothes_restaurant(c, r):
  if (r == "McDonalds"):
    if (c == "shorts and t-shirt"): return 0.9
    elif (c == "dinner suit"):      return 0.1
    elif (c == "tuxedo"):           return 0.1
    elif (c == "tracksuit"):        return 0.5
  elif (r == "L'Escargot"):
    if (c == "shorts and t-shirt"): return 0.001
    elif (c == "dinner suit"):      return 1.0
    elif (c == "tuxedo"):           return 0.8
    elif (c == "tracksuit"):        return 0.1
  elif (r == "Pete's Pork Palace"):
    if (c == "shorts and t-shirt"): return 0.5
    elif (c == "dinner suit"):      return 0.3
    elif (c == "tuxedo"):           return 0.3
    elif (c == "tracksuit"):        return 0.5
```

This expresses the obvious – we should not wear a dinner suit to McDonalds, and we should not wear tracksuit pants

[7] In fact, it is possible to see that our date doesn't actually *care* which restaurant we visit – only how much money we spend.

at a (fancy sounding) French restaurant. Finally, we have the potential between restaurants and movies:

```
def restaurant_movie(r, m):
  if (m == "Star Wars Episode II"):
    return 0.0000001
  if (r == "McDonalds"):
    if (m == "Babe"):                return 0.3
    elif (m == "Supersize Me"):      return 0.01
    elif (m == "Lord of the Rings"): return 0.8
  elif (r == "L'Escargot"):
    if (m == "Babe"):                return 0.4
    elif (m == "Supersize Me"):      return 0.7
    elif (m == "Lord of the Rings"): return 0.6
  elif (r == "Pete's Pork Palace"):
    if (m == "Babe"):                return 0.001
    elif (m == "Supersize Me"):      return 0.6
    elif (m == "Lord of the Rings"): return 0.6
```

Again, this is fairly self explanatory – we should not see a film about a pig with a heart of gold after eating pork, and we should not see an anti-McDonalds film after going to McDonalds. Star Wars Episode II should be avoided at all costs.

Next, we must pass these potential functions as arguments to *discrete.Potentials*, so that they can be used by our algorithms. We also declare that we are using the 'max-product' semiring, and MAP estimation (since we want the optimal date – if we wanted to get dumped, we might use the min-product semiring instead):

```
semiring = semirings.semiring_maxproduct
estimator = discrete.MAPestimate

pot_price_restaurant = discrete.Potentials(price_restaurant, semiring)
pot_clothes_restaurant = discrete.Potentials(clothes_restaurant, semiring)
pot_restaurant_movie = discrete.Potentials(restaurant_movie, semiring)
```

Now we use these potential functions and the nodes we created above to create our three cliques:

```
clique_price_restaurant = cliques.Clique([node_price, node_restaurant],
                                         pot_price_restaurant)
clique_clothes_restaurant = cliques.Clique([node_clothes, node_restaurant],
                                           pot_clothes_restaurant)
clique_restaurant_movie = cliques.Clique([node_restaurant, node_movie],
                                         pot_restaurant_movie)
```

And finally, we are ready to use the junction tree algorithm! We simply pass this set of cliques, along with our chosen estimator to the constructor:

```
jta = junctiontree.JunctionTreeAlgorithm([clique_price_restaurant,
                                          clique_clothes_restaurant,
                                          clique_restaurant_movie],
                                          estimator)
```

Now, to determine the optimal date, we must first pass all messages throughout the tree:

```
jta.propagate()
```

And we can find and print the results as follows:

```
results = jta.findresults([node_price,
                           node_clothes,
                           node_restaurant,
                           node_movie])

print "Price:", results[node_price]
print "Clothes:", results[node_clothes]
print "Restaurant:", results[node_restaurant]
print "Movie:", results[node_movie]
```

Running this program should print out:

```
Price: 70
Clothes: dinner suit
Restaurant: L'Escargot
Movie: Supersize Me
```

Indeed, a wonderful date by any measure.

2.2 Technical Documentation

2.2.1 Overview

In this section, we describe the basic steps required in order for the user to extend the existing models to their own applications. While the algorithms themselves should be generic enough to handle almost any data, the models and semirings may not be. For example, there is currently no model to handle continuous distributions, but it should be easy to write one after having read this section.

2.2.2 Defining your own semirings

The Basics

At its core, a semiring is nothing more than an addition and a multiplication function. Most of the 'standard' addition and multiplication functions are defined in *core.models.semirings*, and most of the 'standard' semirings are defined from these.

All of the basic semirings are defined using the class *semirings.Semiring*, and all new semirings should be an instance of this class. In the simplest case, we need only define an addition function, and a multiplication function, and use them to create an instance of *Semiring*. For example:

```
def sum(a, b):
  return a + b

def product(a, b):
  return a * b

sring = semirings.Semiring("name", sum, product)
```

would create a semiring, which is essentially the same as the sum-product semiring. The name field is used only for error reporting. This semiring can now be used to create a potential function.

Normalisation

The semiring above is not normalisable, since normalisation requires an 'inverse' to be defined on our semiring. This simply means that if this semiring is to be used for a distribution, it will never be normalised, and may cause overflow (note that the distribution class is responsible for normalising the distribution at the appropriate times – see section 2.2.3).

In order for this semiring to be normalisable, we need to define an inverse method. In the case of the sum-product semiring, the inverse is just the reciprocal. That is:

```
def inv(a):
  return 1.0/a
```

That is, if our distribution sums to a, then multiplying the distribution by $1/a$ will result in a distribution that sums to 1. Our new semiring is defined as:

```
sring = semirings.Semiring("name", sum, product, sinv=inv)
```

It is now up to the implementation of *Distribution* to ensure that the distribution is normalised whenever it is appropriate – the user needn't do anything more.

Measuring Difference

In order to evaluate message differences (see section 2.1.4), our semiring must be able to return the difference between two values. In the case of the sum product semiring, this should be something like:

```
def diff(a, b):
  return abs(a - b)
```

This will now be called by the *difference* routine defined by our distribution. Finally, our semiring becomes:

```
sring = semirings.Semiring("name", sum, product, sinv=inv, sdiff=diff)
```

In fact, this is almost exactly the same as the sum-product semiring (*semirings.semiring_sumproduct*), already defined in *core.models.semirings*.

Numpy Semirings

Since the *Semirings* class performs operations on *every* element of a distribution, they are not compatible with numpy (i.e. there would be no performance increase). Therefore, an additional semirings class, *semirings.NumpySemiring* has been defined, and is used by the *discrete_numpy* module.

The principle difference is that we now have a function that sums across rows. This can now be used as part of a marginalisation routine, resulting in a significant performance increase.[8] This function should operate directly on a numpy array. This function could easily be one the *sum*, *max*, or *min* methods that are already defined on numpy arrays.

Also, our *sum* and *product* functions now add and multiply whole distributions. In fact, since the numpy implementations of '+' (or '-') and '*' correspond to elementwise addition and multiplication, we can use the *sum*, *product*, and *diff* functions we defined above.

We still need to define an *inverse* function, and a *sum-row* function. For the sum-product semiring, the *sum-row* function is just:

```
def sum_row(a, row):
  return a.sum(row)
```

Similarly, the inverse function is just:

```
def inv(a):
  return a / a.sum()
```

Now, our numpy semiring can be created using (where 'sum', 'product', and 'diff' are just the functions defined previously):

```
sring = semirings.Semiring("name", sum_row, product, ssum=sum, sinv=inv, sdiff=diff)
```

In reality, the semirings already defined in *core.models.semirings* should already be a fairly conclusive set, and there should rarely be a need to define new ones.

[8] Unfortunately, this method doesn't translate well to the log-domain, so there is currently no log-domain semiring for use with numpy.

2.2.3 Defining your own models

The Basics

The implementation of a 'model' has been defined such that it should be easily extensible to new models. The classes defined in *core.models.interface* contain all of the functions that any user-defined model should implement. Many of these functions aren't implementable by the parent class, so this module should be seen as an interface that the user may implement. Any of the methods that are *not* implemented at this level will raise an error if they are called.

This module contains three classes. Firstly, *BasicNode* should be the parent class for any user-defined node type. Secondly, *BasicDistribution* contains all of the methods required to define a distribution (many of which are not implemented at this level). Finally, *BasicPotentials* is a class that will take a set of nodes, and use a given potential function to actually build a distribution object.

These three classes will be covered separately. Of course, defining an entire distribution is too large a task for us to provide conclusive examples of all functions, so it is probably best to look at the module itself, as well as its derivatives (such as the discrete model, *core.models.discrete*), for more details.

Nodes

As we mentioned in section 2.1.3, a node is just a container for a domain. Its initialiser will presumably take a domain, and it must implement a function to return this domain.

In the class itself (*core.models.interface.BasicNode*), the methods to be implemented have been split into two categories: 'Mandatory' methods, for those methods that are required by loopy-belief propagation or the junction-tree algorithm, and 'Optional' methods, for those methods that are only required by 'From Fields to Trees', 'EM for HMMs', or some other algorithm. In this documentation, we shall describe only the mandatory methods in full detail, but the user should be aware that if they want to use their distribution with one of the other algorithms, they will need to implement (at least some of) these optional methods, and may need to refer to the module itself. In addition, some of the basic getter/setter methods have been ignored – these are largely self explanatory, but please refer to the module itself if more description is required.

Mandatory Methods:

initialiser Should take a domain, and probably a name (for printing/debugging).

getdomain Returns this node's domain. This will be needed when we define our own *Potentials* class (section 2.2.3), which will need to know the domains of all nodes in a clique in order to determine that clique's distribution.

isbeliefnode Returns 'True' if and only if this node is a belief node (in the discrete case, for example, this just returns 'True' if the domain contains only a single element). While this method is mandatory, a minimal implementation may just return 'False' at all times, if belief nodes should not receive any special treatment in a particular model.

Optional Methods:

fixvalue Sets this node's domain to a fixed value – hence making it a belief node. This is needed by the 'From Fields to Trees' algorithm.

randomvalue Select a value randomly from our domain.

unfixvalue Set this node *not* to be a belief node anymore. For this to be implemented, 'fixvalue' may have to store what the domain was before the value was 'fixed'.

Many of the above methods are simple enough that they are implemented at the top level. Therefore, there should little work involved in defining a new node type.

Distributions

In essence, a 'Distribution' requires the implementation of two functions – marginalisation and multiplication. Although there are a number of other methods, most of them should be trivial.

Mandatory Methods:

initialiser Should take a set of nodes, and their distribution (however it is internally represented by our model). Note that the distribution is only *stored* by this class – the class responsible for actually *building* the distribution is the *Potentials* class, defined below. The initialiser should also take a semiring, which may be used to perform marginalisation and multiplication.

distributionat This method should take some assignment to all of its nodes, and simply evaluate the distribution at this point. This function will probably be called by the estimator (we will show how to build these in section 2.2.3). If you wish to implement your own estimator, and it doesn't require such a method, then *distributionat* does not need to be implemented – it will not be called elsewhere. This may require further explanation, so we will provide some specific examples in section 2.2.3.

marginalise This method should take a set of nodes (a subset of this distribution's nodes), and return a new distribution, corresponding to this distribution's marginal with respect to those nodes. This will be described in more detail below.

multiplymarginal Like marginalise, this method should also take some subset of this distribution's nodes, and should return a new distribution, corresponding to this distribution, multiplied by that marginal.

Optional Methods:

add This should simply 'add' two distributions (in accordance with our semiring). Each of the distributions should be over exactly the same set of nodes. This is needed by the 'From Fields to Trees' algorithm in order to compute the Rao-Blackwellised estimate.

bias This should take another distribution (say $\mathbf{x}$), and a bias term (say, β) and return this distribution, plus $\beta\mathbf{x}$. This is sometimes used as an alternate update equation for loopy belief-propagation.

difference This should take two distributions, and return some measure of the difference between them. This difference may then be used as a stopping criteria for loopy belief-propagation (see section 2.1.4).

dividemarginal This is like *multiplymarginal*, except that it performs division instead of multiplication. This is needed by the EM algorithm.

normalise Normalise this distribution.

Potentials

The *Potentials* class is fairly straightforward – it needs only one method, *finddistribution*, which should take a set of nodes, and return a *Distribution* object, corresponding to the joint distribution for those nodes (using a potential function passed to its initialiser).[9]

The reason we implement the *finddistribution* function, rather than just putting it in the initialiser, is that some algorithms may require that we change the domain of some nodes (such as 'From Fields to Trees'), and some others may even require that we change the potential functions themselves (such as the EM algorithm). Therefore, these algorithms will just call *finddistribution* every time a change is made.

If this is still confusing, see the examples in section 2.2.3, or see the module itself.

[9] By necessity, this class is somewhat exposed to the internal representation of our distribution

Estimators

An 'estimator' should simply be a method which takes a distribution, and returns some assignment to that distribution's nodes. While it may appear that this is impossible without exposing the internal representation of a distribution, this is not necessarily the case. For example, the estimators defined in *core.models.discrete* need only to call *Distribution.getnodes* and *Distribution.distributionat* (defined in section 2.2.3) in order to make a MAP estimate, or take a sample from a distribution. Of course, for a different model, taking an estimate may be more complicated, and may require the user to implement additional helper functions in their *Distribution* class.

Examples

If the above methods are slightly confusing, then they may be made more clear with some examples. For instance, suppose we have defined some model, in which nodes are discrete valued, and distributions are internally represented using an array.[10] Then our nodes and potential function may be defined (as in section 2.1.3) as follows:

```
n1 = model.Node(["happy", "sad"])
n2 = model.Node(["wet", "dry"])

def pfunction(x, y):
  return {("happy", "wet"):0.2, ("happy", "dry"):0.4,
          ("sad", "wet"):0.25, ("sad", "dry"):0.15}[(x,y)]

pot = model.Potentials(pfunction, semirings.semiring_sumproduct)
```

Then if we were to run:

```
dist = pot.finddisttribution([n1,n2])
```

We will get a distribution whose internal representation is something like:

```
           "wet"   "dry"
"happy" [[ 0.2,    0.4  ],
"sad"    [ 0.25,   0.15 ]]
```

Now, if we use the *distributionat* method (section 2.2.3), and make the call:

```
dist.distributionat("happy", "dry")
```

then it should return a value of '0.4'. Estimates should be taken from this distribution using calls such as:

```
discrete.MAPestimate(dist)
```

which would return ('happy', 'dry') (as that is the configuration with the highest probability). To marginalise this distribution with respect to *n1*, we would call:

[10] This is exactly the case with the numpy implementation of the discrete model.

```
dist2 = dist.marginalise([n1])
```

The internal representation of this distribution would now be something like:

```
"happy" [ 0.6,
"sad"     0.4 ]
```

Finally, suppose we wish to multiply *dist* by *dist2*. We would call:

```
dist3 = dist.multiplymarginal(dist2)
```

Then the internal representation of *dist3* will be something like:

```
           "wet"    "dry"
"happy" [[ 0.230,   0.462 ],
"sad"    [ 0.192,   0.115 ]]
```

Note that the above distribution has been normalised. This can be equivalently expressed as:

```
           "wet"       "dry"
"happy" [[ 0.2*0.6,    0.4*0.6  ],
"sad"    [ 0.25*0.4,   0.15*0.4 ]] / 0.52
```

which should clarify what is happening. This is *not* the same as *dist2.multiplymarginal(dist)*, which is actually not even defined.

Of course, it may be the case that the user never actually needs to call these functions themselves – they will all be invoked at the appropriate times by other algorithms.

2.3 Appendix

2.3.1 Notes on the Code

Should this documentation be insufficient, the obvious next port-of-call is the code itself. Although this code is heavily commented, there are a few notes on style etc. that should be observed before trying to read it.

Member Variables

All member variables in the code should be considered private (although this is unenforceable in Python). They should be accessed only using the 'getter' methods that correspond to them. The 'getter' methods provided should correspond to exactly those variables that *should* be visible by the user.

'Get' and 'Find' Methods

Although there may appear to be some inconsistency in the naming of methods such as *getcliques* and *findmarginaldistributions*, there is actually a reason for naming them differently – I have adopted the convention that a 'get' method only returns static data (it may also call other 'get' methods). Hence calling such a method repeatedly will not be a significant computation burden. 'Find' methods, on the other hand, may need to perform substantial calculations before they return their results – therefore it is important to cache the values they return, if they are to be used repeatedly.

CHAPTER

THREE

Kernel Methods for Estimation

3.1 Introduction

This module implements several methods for estimation: support vector classification, support vector regression, gaussian process classification, gaussian process regression, quantile estimation and novelty detection.

At present it is using fairly basic optimization techniques, i.e. Newton solvers with conjugate gradient, reduced rank expansions and interior point solvers to address the optimization problems arising in this context. For details on the techniques described in this chapter see Gibbs and Mackay (1997); Le et al. (2005); Schölkopf and Smola (2002); Schölkopf et al. (1999); Schölkopf et al. (2001); Takeuchi et al. (2006); Vapnik (1995) and references therein.

3.1.1 Known Issues

- The algorithms at present do not scale well beyond 5000 observations. This is because by default they do not make use of reduced rank expansions and similar methods for efficiency. This deficiency will be addressed in subsequent releases.
- Range checking in the algorithms is only sporadically implemented. That is, if you set the wrong parameters, the method may fail for no obvious reason. This will be added.
- Model selection is somewhat rudimentary at the moment. Codes for automatic adaptation for transduction and validation techniques will follow.

3.1.2 File Hierarchy

The files used in this document are all contained in *estimation* of Elefant. The hierarchy of the files contained therein is as follows:

estimation:

estimate.py	Front end user interface for estimation (section **??**)
evaluate.py	Front end user interface for evaluation (section **??**)
estimation/svm/:	
classification.py	SV Classification classes (section 3.7.1)
regression.py	SV Regression classes (section 3.7.2)
estimation/gp:	
classification.py	GP Classification classes (section 3.8.3)
regression.py	GP Regression classes (section 3.8.1)
heteroscedastic_regression.py	Heteroscedastic GP Regression classes (section 3.8.2)
quantile.py	Nonparametric quantile estimation (section 3.9)
novelty.py	Novelty detection (section 3.10)

3.1.3 Dependencies

scipy is needed for optimization.

matplotlib is required for users wishing to execute test programs and produce plots.

3.2 Quick-Start Guide

To use the kernel package, issuse the following command in your terminal (assume that you already in the estimation directory)

```
python estimate.py
```

This will display a help message on how to use a different methods in combination with different types of kernels. At the end of the message, there are simple examples on how to use this program.

After successfully building your models, you could type the following command

```
python evaluate.py
```

to see how to evaluate your model.

Note that the current accepted file formats are comma separated and tab separated (the svmlight format is currently not supported).

The package also contains **data** folder which stores some simple files.

The following sections describe the details of the classes. The descriptions are more suitable for the developers.

3.3 Basic Standard

Most classes in the **estimation** package have a simple interface. There are three main methods. the first method is the constructor where the user needs to supply the parameters for the models, most parameters are set as default values. The second method is *Train* where the user is required to supply training data and labels. The last method is *Test* where user is required to supply the test data.

Note: exceptions are transductive gaussian process classification and heteroscedastic guassian process regression (See below).

3.4 Test programs

For each file described below, simply issue the following command

```
python {program}.py
```

to run the program tests. The programs should generate random example datasets, perform estimation and display a nice picture.

3.5 Examples

The folder **estimation/examples/testionosphere.py** contains an example of how to train two classifiers: CSVC and NuSVC. The descriptions of the classifiers please see the details below. For now, assume that the they are simply two support vector classifiers.

Most of the code contains random splitting of the ionosphere and selecting the model parameters. Let's analyze the code.

The following line imports the two classifiers:

```
from elefant.estimation.svm.classification import CSVC, NuSVC
```

We choose to use the Gaussian RBF kernel in this example:

```
from elefant.kernels.vector import CGaussKernel
```

We then try to load the ionosphere data by using the command

```
data = file2np('ionosphere.pyformat')
```

file2n is a function defined in *reader.py* to read space-separated file format.

```
x = data[:,:n-1]
y = data[:,n-1:n]
```

The above two commands create the x matrix and the y vector from *data*. If there are m data points, and each has n dimensions then the matrix x should have m rows and n columns. This means in Python data is stored row by row (row-major).

Then we start splitting the dataset into two parts, *xtrain* and *ytrain* for training; *xtest* and *ytest* for test.

The next step is to loop over the space of model parameters to find the best parameters that give best results on the test set (notice, that this is not cross validation: there is no validation set).

We first create and train the NuSVC the following lines are essential

```
kernel = CGaussKernel(omega)
q = NuSVC(nu, C, kernel)
q.Train(xtrain,ytrain)
mean = sign(q.Test(xtest))
```

The first line creates the kernel object of Elefant. The second line creates the object of type *NuSVC* and sets up all relevant parameters. The next line supplies *xtrain* and *ytrain* to *Train* the model. Finally, we can perform prediction by supplying *xtest* to *Test*. We generally have to use the sign function to get back the correct labels.

Note that it is worth emphasizing that this is the typical interface of classifiers and regressors: 1) the constructor needs the relevant parameters 2) the *Train* method needs *xtrain* and *ytrain* 3) the *Test* method needs *xtest*. For classifiers in **estimation/svm/classification.py** we need the sign function in **utils/vector.py** to get back the correct labels.

3.6 API Reference

3.7 Support Vector Machines

3.7.1 Support Vector Classification

Path: **estimation/svm/classification.py**
Classes: CSVC, NuSVC

CSVC

is an implementation of the well-known soft margin C-SVC (C Support Vector Classification) Cortes and Vapnik (1995).

NuSVC

contains an implementation of the ν-SVC (New Support Vector Classification) Schölkopf et al. (2000).

3.7.2 Support Vector Regression

Path: **estimation/svm/regression.py**
Classes: EpsilonSVR, NuSVR, LaplacianSVR

EpsilonSVR

contains an implementation of the ϵ-insensitive loss support vector regression Vapnik (1995), where ϵ is the size of the tube.

NuSVR

contains an implementation of the ν-SVR (New Support Vector Regression) Schölkopf et al. (2000).

LaplacianSVR

contains an implementation of the Laplacian loss Support Vector Regression Schölkopf and Smola (2002).

3.8 Gaussian Processes

3.8.1 Gaussian Process Regression

Path: **estimation/gp/regression.py**
Classes: Simple

Simple

contains an implementation of the Gaussian Process Regression Rasmussen and Williams (2006). We basically use cholesky decomposition to resolve the inverse of the covariance matrix.

3.8.2 Heteroscedastic Gaussian Process Regression

Path: **estimation/gp/heteroscedastic_regression.py**
Classes: Heteroscedastic

Heteroscedastic

contains an implementation of the Heteroscedastic Gaussian Process Regression Le et al. (2005). Unlike the standard Gaussian Process Regression, the heteroscedastic Gaussian Process Regression makes no assumption about the uniformity of the noise level. The model enables the users to learn the mean and the covariance at the same time. Users are required to supply two kernels and two regularization constants to learn the mean and the covariance. Incomplete cholesky factorization can be used to deal with high storage cost. Note: This class does not conform the basic standard as stated above.

3.8.3 Gaussian Process Classification

Path: **estimation/gp/classification.py**
Classes: Multiclass, ExtremeMulticlass, TransductiveMulticlass, InverseTransductiveMulticlass

Multiclass

contains an implementation of the standard multiclass Gaussian Process Classification Rasmussen and Williams (2006).

ExtrememMulticlass

contains an implementation of transductive multiclass classification with extreme moment matching. WARNING: this is currently experimental.

TransductiveMulticlass

contains an implementation of transductive multiclass Gaussian Process classification with moment matchingGärtner et al. (2006). It generally assumes that the fractions of positive and negative examples on training and test data match up approximately. Users are require to supply a parameter telling how much the two fractions match up.

Note: This class does not conform the basic standard above (Section 3.3). Since this is transductive, the users are required to supply the training + test data at the same time. Training and test data are supply in a form of a big matrix, training and initial guess of test predictions should also be supplied in a big vector. It is also required to input the position where training and test data are split in the input data. The program will output the prediction in terms of class labels, to get back the probability users should access the attribute *pi*.

InverseTransductiveMulticlass

contains an implementation of transductive multiclass Gaussian Process classification with moment matchingGärtner et al. (2006) in inverse form. The inverse form is particularly useful when the computation of the kernel matrix requires a matrix inverse (e.g. graph kernels). The users are required to supply the non-inverse graph kernels.

Note: This class does not conform the basic standard above (Section 3.3). Since this is transductive, the users are required to supply the training + test data at the same time. Training and test data are supply in a form of a big matrix, training and initial guess of test predictions should also be supplied in a big vector. It is also required to input the position where training and test data are split in the input data. The program will output the prediction in terms of class labels, to get back the probability users should access the attribute *pi*.

3.9 Nonparmetric Quantile Estimation

Path: **estimation/quantile.py**
Classes: Quantile

3.9.1 Quantile

contains an implementation of the nonparametric quantile estimation Takeuchi et al. (2006). It deals with a non-standard loss function called pinball loss. This estimator is useful for estimating the quantile functions such as the median.

3.10 Novelty

Path: **estimation/novelty**
Classes: OneClass

3.10.1 OneClass

contains an implementation of the one-class SVMs Schölkopf et al. (2001). This algorithm is useful for outlier or novelty detection. One class SVMs are also known as Single class SVMs.

BIBLIOGRAPHY

C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 20(3):273–297, 1995.

T. Gärtner, Q.V. Le, S. Burton, A. J. Smola, and S. V. N. Vishwanathan. Large-scale multiclass transduction. In Yair Weiss, Bernhard Schölkopf, and John Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 411 – 418, Cambride, MA, 2006. MIT Press.

Mark Gibbs and David J. C. Mackay. Efficient implementation of Gaussian processes. Technical report, Cavendish Laboratory, Cambridge, UK, 1997. available at http://wol.ra.phy.cam.ac.uk/mng10/GP/.

Q. V. Le, A. J. Smola, and S. Canu. Heteroscedastic gaussian process regression. In *Proc. Intl. Conf. Machine Learning*, 2005.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

B. Schölkopf, P. L. Bartlett, A. J. Smola, and R. C. Williamson. Shrinking the tube: a new support vector regression algorithm. In M. S. Kearns, S. A. Solla, and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11*, pages 330–336, Cambridge, MA, 1999. MIT Press.

B. Schölkopf, A. J. Smola, R. C. Williamson, and P. L. Bartlett. New support vector algorithms. *Neural Computation*, 12:1207–1245, 2000.

B. Schölkopf, J. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

I. Takeuchi, Q.V. Le, T. Sears, and A.J. Smola. Nonparametric quantile estimation. *Journal of Machine Learning Research*, 2006. To appear and available at http://sml.nicta.com.au/∼quocle.

V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

CHAPTER

FOUR

Hilbert-Schmidt Independence Criterion and Backward Elimination for Feature selection

4.1 Hilbert-Schmidt Independence Criterion

4.1.1 Mathematical definition

Hilbert-Schmidt Independence Criterion (HSIC) is defined as the square of the Hilbert-Schmidt norm of the cross-covariance operator defined in Gretton et al. (2005). It can be used to measure the dependence between the data and the labels. Given a data set x and the labels y, the biased empirical HSIC in terms of the kernel matrix, G, on the data and the kernel matrix, L, on the labels is:

$$\mathrm{HSIC} = \frac{1}{(\mathrm{m}-1)^2}\mathrm{Tr}(\mathrm{GHLH}) \tag{4.1}$$

where $\mathrm{Tr}(\star)$ computes the trace of a matrix and m is the number of data points. The entries in the centering matrix H are defined as $(\mathrm{H})_{\mathrm{ij}} = \delta_{\mathrm{ij}} - \mathrm{m}^{-1}$. The unbiased HSIC is:

$$\mathrm{HSIC} = \frac{1}{\mathrm{m}(\mathrm{m}-3)}\left(\mathrm{Tr}(\mathrm{GL}) + \frac{1}{(\mathrm{m}-1)(\mathrm{m}-2)}1^\top \mathrm{G}11^\top \mathrm{L}1 - \frac{2}{\mathrm{m}-2}1^\top \mathrm{GL}1\right) \tag{4.2}$$

where 1 is the a vector of all ones with corresponding length, and the diagonal entries of G and L are set to zeros in this case.

4.1.2 Interface for HSIC computation

All HSIC related computations are implemented in the class CHSIC. There are four major functions to compute the biased and the unbiased HSIC in a normal way or in a fast way. These functions are:

`BiasedHSIC(x, y, kernelx, kernely)` Compute

> the biased HSIC with the data x and the labels y. kernelx and kernely are the kernel classes that operate on x and y respectively. The default kernel for the data and the kernel are both linear kernel (CLinearKernel). The data type of x and y can be anything, but the users have to insure that the provided kernel can operate their corresponding data.

`UnBiasedHSIC(x, y, kernelx, kernely)` Compute

the unbiased HSIC with the data x and the labels y. The default kernel for the data and the kernel are both linear kernel.

`BiasedHSICFast(G, HLH)` Compute

biased HSIC with precomputed kernel matrix, G, on the data and the centered kernel matrix, HLH, on the labels.

`UnbiasedHSICFast(G, L, sL, ssL)` Compute

unbiased HSIC with precomputed kernel matrix, G, on the data and the kernel matrix, L, on the labels. Two additional arguments are needed, the vector sL (each entry of sL is the sum of one row of L) and the scalar ssL (the sum of all entries in L).

Note that in the interface, all matrices are of type numpy.array.

4.1.3 Additional function for HSIC

Additional functions are used to optimize HSIC with respect to the kernel parameters. Currently, they are mainly used to optimize the HSIC with respect to the scale parameter of a Gauss kernel, $\exp(-\text{scale}\|\text{x1}_\text{i} - \text{x2}_\text{j}\|^2)$ or a Laplace Kernel $\exp(-\text{scale}\|\text{x1}_\text{i} - \text{x2}_\text{j}\|)$ when using them for feature selection. The corresponding optimization problems is of the form:

$$\min_{\text{scale}} \; -\text{HSIC}(\text{scale}) \tag{4.3}$$

The interface of the four related methods are:

`ObjBiasedHSIC(param, x, kernelx, HLH)` Use the biased HSIC as an optimization objective. This function simply returns the negative of HSIC computed on the data x.

`GradBiasedHSIC(param, x, kernelx, HLH)` Return the gradient of the negative of the biased HSIC with respect to the kernel parameters param. For current version, param is a scalar.

`ObjUnBiasedHSIC(param, x, kernelx, HLH)` Use the unbiased HSIC as an optimization objective. This function simply returns the negative of HSIC computed on the data x.

`GradUnBiasedHSIC(param, x, kernelx, HLH)` Return the gradient of the negative of the unbiased HSIC with respect to the kernel parameters param.

4.2 Backward Elimination for Feature selection

4.2.1 The algorithm

Using a universal kernel such as the Gauss kernel or the Laplace kernel, HSIC is zero if the data and class labels are independent; For feature selection, clearly we want to reach the opposite result, namely strong dependence between expression levels and class labels. Hence we try to select features that maximize HSIC.

Having defined our feature selection criterion, we now describe an algorithm that conducts feature selection on the basis of this dependence measure. Using HSIC, we can perform both forward and backward selection of the features. In particular, when we use a linear kernel on the data and labels, forward selection and backward selection are equivalent. However, although forward selection is computationally more efficient, backward elimination in general yields better features, since the quality of the features is assessed within the context of all other features. Hence we present the backward elimination (BA) version of our algorithm here.

Our feature selection algorithm (BAHSIC) appends the features from $\mathcal{S}$ to the end of a list $\mathcal{S}^\dagger$ so that the elements towards the end of $\mathcal{S}^\dagger$ have higher relevance to the learning task. To select t features, we can simply take the last t elements from $\mathcal{S}^\dagger$. The overall algorithm produces $\mathcal{S}^\dagger$ using a backward elimination procedure. It proceeds recursively, eliminating the least relevant features from $\mathcal{S}$ and adding them to the end of $\mathcal{S}^\dagger$ in each iteration.

Algorithm 1 Feature Selection via Backward Elimination

Input: The full set of features $\mathcal{S}$
Output: An ordered set of features $\mathcal{S}^\dagger$

1: $\mathcal{S}^\dagger \leftarrow \varnothing$
2: **repeat**
3: $\quad \sigma_0 \leftarrow \arg\max_\sigma \ \mathrm{HSIC}(\sigma, \mathcal{S}), \ \ \sigma \in \Xi$
4: $\quad i \leftarrow \arg\max_i \ \mathrm{HISC}(\sigma_0, \mathcal{S} \setminus \{i\}), \ \ i \in \mathcal{S}$
5: $\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{i\}$
6: $\quad \mathcal{S}^\dagger \leftarrow \mathcal{S}^\dagger \cup \{i\}$
7: **until** $\mathcal{S} = \varnothing$

Step 3 of the algorithm optimises over all possible choices of kernel parameters in the set Ξ. Note that Ξ is chosen such that the kernels are bounded. If we have no prior knowledge regarding the nature of the nonlinearity in the data, then optimising over Ξ is essential: it allows us to adapt to the scale of the nonlinearity present in the (feature-reduced) data. If we have prior knowledge about the type of nonlinearity, we can use a kernel with fixed parameters for BAHSIC. In this case, step 3 can be omitted since there will be no parameter to tune. For faster elimination of features, we can choose a group of features at step 4 and delete them in one shot at step 5.

Note that by choosing different kernels, BAHSIC can be applied to binary, multiclass and regression problems. More more information, see reference Song et al. (2006).

4.2.2 Interface to BAHSIC

Two versions of the BAHSIC, with and without the optimization over the parameters, are implemented as methods of the class CBAHSIC. The interface to these methods are:

```
BAHSICRaw(x, y, kernelx, kernely, flg3, flg4)
```
BAHSIC

without the optimization over the kernel parameters. This is useful when using kernels of fixed parameters or without parameters for feature selection. In this case, we instill our prior knowledge into the feature selection and desire only features that are detectable by the given kernels. For instance, applying polynomial kernel of fixed degree or linear kernel on the data for the feature selection. The two additional arguments, flg3 and flg4, both scalars, controls the numbers of desired features and the proportion of feature eliminated in each iteration

respectively. Note that flg3 can not exceed the maximum number of features available, and flg4 is a number in (0,1]. Note that for feature selection, usually a linear kernel is applied on the labels.

`BAHSICOpt(x, y, kernelx, kernely, flg3, flg4)` BAHSIC

with the optimization over the kernel parameters. This is particularly useful when we don't have any prior knowledge on the desired features from the data. In this case, usually a universal kernel, such as Gauss kernel and Laplace kernel, is applied and we try to detect features of any kinds. Therefore, we need to optimize over the scale parameters as indicated in the BAHSIC algorithm. Similarly flg3 and flg4 controls the numbers of desired features and the proportion of feature eliminated in each iteration respectively. Note that even in this version with optimization, usually a linear kernel is also applied on the labels. This method uses the fmin_cg routine from scipy for the optimization. However, there seems to be some speed and accuracy issues in scipy optimization routine. This degrades the performance of the current version of BAHSICOpt.

For more information, see the testing examples in this package.

BIBLIOGRAPHY

A. Gretton, O. Bousquet, A.J. Smola, and B. Schölkopf. Measuring statistical dependence with Hilbert-Schmidt norms. In S. Jain and W.-S. Lee, editors, *Proceedings Algorithmic Learning Theory*, 2005.

Le Song, Justin Bedo, Karsten M. Borgwardt, Arthur Gretton, and Alex Smola. The bahsic family of gene selection algorithms. *submitted*, 2006.

CHAPTER

FIVE

Kernels

5.1 Overview

This package provides fast computation of the kernel matrices and various fast manipulation of kernel matrices. It builds on the matrix and vector operation of Numpy. The peak performance of the kernel computation is typically higher than the same operation implemented in Matlab. Furthermore, to allow easy extension of the package, all kernels in this package are object-oriented. A common interface for kernels is defined. Overloading the methods in the interface allows the users to customize for their own kernel classes. In this chapter, we will focus on the common interface of the kernels, several key derived classes, and the implementation tricks that lead to the fast computations.

5.2 Hierarchy of current kernel classes

Currently, the classes in the kernel package have the hierarchy as depicted in Figure 5.1.

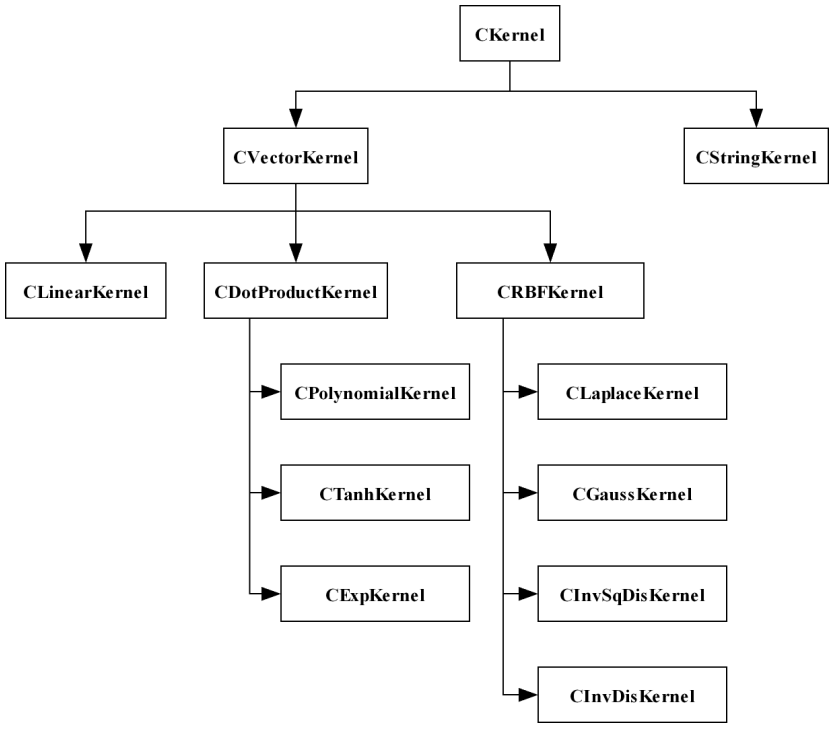

Figure 5.1: Hierarchy of kernels that operate on vectors currently implemented in the kernel package.

5.3 Notations

To facilitate our later explanation, we first define some notations here. Let x1 and x2 be data sets, The number of data points in x1 and x2 are m1 and m2 respectively. Let $K(x1_i, x2_j)$ denote the kernel between two data points $\mathrm{x1_i} \in \mathrm{x1}$ and $\mathrm{x2_j} \in \mathrm{x2}$. Let G denote the Gram matrix (or kernel matrix) and $(\mathrm{G})_{\mathrm{ij}}$ denote the ij-th entry in G. Let y1 and y2 the vectors of labels. Furthermore let alpha be the vector of coefficients (eg. estimated by SVM or SVR). Note that for this chapter, a matrix of vectorial data consists of rows of vectors with each row a data point. For a single vector, it is represented as a matrix with a single column (ie. its shape is $(\star, 1)$). Let v denote a vector and $(\mathrm{v})_{\mathrm{i}}$ be the i-th entry in the vector. Let $\otimes$ denote the tensor product (elementwise product or Hadamard product) between two matrices.

5.4 Common interface

The common interface called CKernel is defined in elefant.generic. Six key methods are defined in this interface. For more information on the usage of this operations in kernel methods, see reference Schölkopf and Smola (2001). The programming interface of this methods are described below.

`__init__(blocksize=128)` The constructor takes one parameter blocksize. During the manipulation of the kernel matrix, the data sets can be partitioned into subsets of the size determined by the parameter blocksize. Thus the entries in the resulting matrices (or vectors) are computed block by block. This makes use of the caching mechanism of the OS and speeds up the computation. The default subset size is 128 and it can be changed to match different machine and systems.

`K(x1, x2)` This method computes the kernel value between two data points x1 and x2. Note that the data type of x1 and x2 have to be the same, but the exact data type is not specified by this common interface. Depending on applications, the data type can be of any type, such as vector, string or graphs. It returns a scalar.

`Dot(x1, x2, index1=None, index2=None, output=None)` This method computes the kernel matrix, G, for two sets of data, x1 and x2. The exact data type of each data point is not specified in this interface, as long as they are of the same type. Each entry in G corresponds to the kernel value $\mathrm{K}(\mathrm{x1}, \mathrm{x2})$ for two data points $\mathrm{x1_i} \in \mathrm{x1}$ and $\mathrm{x2_j} \in \mathrm{x2}$, ie. $(\mathrm{G})_{\mathrm{ij}} = \mathrm{K}(\mathrm{x1_i}, \mathrm{x2_j})$. The arguments, index1 and index2, are the indices into data set x1 and x2 respectively. When index1 (or index2) is provided, only a subset of the data from x1 (x2) indexed by index1 (index2) is used to compute the kernel matrix. Therefore the resulting kernel matrix is of size $\mathrm{len}(\mathrm{index1}) \times \mathrm{len}(\mathrm{index2})$. When either index1 or index2 is not provided, the default is to compute the kernel matrix using the corresponding full set of data. The argument output is used to specify a explicit buffer for storing the resulting kernel matrix. If output is not provided, a new space is allocated to store the kernel matrix. By default, it returns a numpy.array of size (m1, m2).

`Expand(x1, x2, alpha, index1=None, index2=None, output=None)` This method computes the multiplication of the kernel matrix, G, with the vector of coefficients, alpha. The argument alpha is a vector of size (1, m2) (m2 is the number of data points in data set x2). Mathematically, the resulting vector v is defined as $\mathrm{v} = \mathrm{G}(\mathrm{alpha})$, ie. $(\mathrm{v})_{\mathrm{i}} = \sum_{\mathrm{j}=1}^{\mathrm{m2}} (\mathrm{G})_{\mathrm{ij}} (\mathrm{alpha})_{\mathrm{j}} = \sum_{\mathrm{j}=1}^{\mathrm{m2}} \mathrm{K}(\mathrm{x1_i}, \mathrm{x2_j})(\mathrm{alpha})_{\mathrm{j}}$. By default, it returns a numpy.array of size (m1, 1).

`Tensor(x1, y1, x2, y2, index1=None, index2=None, output=None)` This method computes the tensor product between the kernel matrix, G, and the matrix $\mathrm{y1y2}^{\top}$. The two

arguments, y1 and y2, are the label vectors for data set x1 and x2, and they are of the size (1, m1) and (1, m2) respectively. Mathematically, the returning matrix is defined as $\mathrm{M} = \mathrm{G} \otimes (\mathrm{y1y2}^\top)$, ie. $(\mathrm{M})_{\mathrm{ij}} = (\mathrm{G})_{\mathrm{ij}}(\mathrm{y1})_{\mathrm{i}}(\mathrm{y2})_{\mathrm{j}} = \mathrm{K}(\mathrm{x1}_{\mathrm{i}}, \mathrm{x2}_{\mathrm{j}})(\mathrm{y1})_{\mathrm{i}}(\mathrm{y2})_{\mathrm{j}}$. By default, it returns a numpy.array of size (m1, m2).

`TensorExpand(x1, y1, x2, y2, alpha, index1=None, index2=None, output=None)` This

method first computes the tensor product between the kernel matrix, G, and the matrix $\mathrm{y1y2}^\top$, and then multiply this intermediate result by the vector of coefficients alpha. Mathematically, the returning vector, v, is defined as $\mathrm{v} = (\mathrm{G} \otimes (\mathrm{y1y2}^\top))(\mathrm{alpha})$, ie. $(\mathrm{v})_{\mathrm{i}} = \sum_{\mathrm{j}=1}^{\mathrm{m2}} (\mathrm{G})_{\mathrm{ij}}(\mathrm{y1})_{\mathrm{i}}(\mathrm{y2})_{\mathrm{j}}(\mathrm{alpha})_{\mathrm{j}} = \sum_{\mathrm{j}=1}^{\mathrm{m}} \mathrm{K}(\mathrm{x1}_{\mathrm{i}}, \mathrm{x2}_{\mathrm{j}})(\mathrm{y1})_{\mathrm{i}}(\mathrm{y2})_{\mathrm{j}}(\mathrm{alpha})_{\mathrm{j}}$. By default, it returns a numpy.array of size (m1, 1).

`Remember(x1)` Remember

data set x1 by storing the preprocessing results in the cache, so that it speeds up later computations. The preprocessing can vary from kernel to kernel.

`Forget(x1)` Remove a

remembered data set x1 from the cache.

To design a specific kernel, simply overload these methods. Note that the generic kernel class CKernel itself should never be instantiated. When overloading the methods, insure the following consistencies:

`x1, x2` Insure they

are of the same data type.

`y1, y2` Insure they

are of the same data type

`alpha` alpha be a

numpy.array of size (m2,1)

`index1, index2` Insure in-

dex1 and index2 are of the same type, and their type are either list or one dimensional numpy.array.

`output` Insure that

the output buffer is larger than the space required to hold all resulting entries of the matrices or vectors.

5.5 Kernels on vectorial data

5.5.1 Data type

All kernels operated on vectorial data are defined in the package elefant.kernels.vector. These classes are derived from CKernel. The interfaces are instantiated to take the following data type:

`x1, x2` The data x1

and x2 are two dimensional numpy.array.

5.5.2 Details of the kernels

The class CVectorKernel is the abstract class for all kernels that operate on vectorial data. It has three main branches of descendants:

`CLinearKernel` This kernel

computes conventional inner product (vanilla dot) on two data points, ie. $\mathrm{K}(\mathrm{x1_i}, \mathrm{x2_j}) = \langle \mathrm{x1_i}, \mathrm{x2_j} \rangle$. This kernel is singled out as a separate class because the manipulation of the resulting kernel matrix can be computed efficiently by blocking and clever bracketing.

`CDotProductKernel` This is an

abstract class. All kernels that are functions of the inner product, ie. $\mathrm{K}(\mathrm{x1_i}, \mathrm{x2_j}) = \mathrm{Kappa}(\langle \mathrm{x1_i}, \mathrm{x2_j} \rangle)$ derive from this class. Currently there are three specific kernels derived from this class. They are different in their constructors and the Kappa function.

`CPolynomialKernel`

`Kappa(x)` Given a

numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \langle \mathrm{x1_i}, \mathrm{x2_j} \rangle$, it computes the polynomial kernel of the form:

$$\mathrm{Kappa}(\mathrm{x}) = (\mathrm{scale} \cdot \mathrm{x} + \mathrm{offset})^{\mathrm{degree}}. \tag{5.1}$$

`__init__(degree=2, offset=1.0, scale=1.0, blocksize=128)` The three

parameters, degree, offset and scale, are all scalars. The default constructor creates polynomial kernel of degree of 2.

`CTahnKernel`

`Kappa(x)` Given a

numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \langle \mathrm{x1_i}, \mathrm{x2_j} \rangle$, it computes the Tahn kernel of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \tanh(\mathrm{scale} \cdot \mathrm{x} + \mathrm{offset}). \tag{5.2}$$

`__init__(offset=1.0, scale=1.0, blocksize=128)` The two

parameters, offset and scale, are both scalars.

`CExpKernel`

`Kappa(x)` Given a

numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \langle \mathrm{x1_i}, \mathrm{x2_j} \rangle$, it computes the Exponential kernel of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \exp(\mathrm{scale} \cdot \mathrm{x} + \mathrm{offset}). \tag{5.3}$$

`__init__(offset=1.0, scale=1.0, blocksize=128)` The two

parameters, offset and scale, are both scalars.

`CRBFKernel`

This is an abstract class. All kernels that are functions of the distance, ie. $\mathrm{K}(\mathrm{x1_i}, \mathrm{x2_j}) = \mathrm{Kappa}(\|\mathrm{x1_i} - \mathrm{x2_j}\|^2)$, derive from this class. Currently there are four specific kernels derived from this class. They are different in their constructors and the Kappa function.

`CLaplaceKernel`

`Kappa(x)` Given a numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \|\mathrm{x1_i} - \mathrm{x2_j}\|^2$, it computes the Laplace kernel of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \exp(-\mathrm{scale} \cdot \sqrt{\mathrm{x}}) \tag{5.4}$$

`__init__(scale=1.0, blocksize=128)` The parameter, scale, is a scalar.

`CGaussKernel`

`Kappa(x)` Given a numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \|\mathrm{x1_i} - \mathrm{x2_j}\|^2$, it computes the Gauss kernel of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \exp(-\mathrm{scale} \cdot \mathrm{x}) \tag{5.5}$$

`__init__(scale=1.0, blocksize=128)` The parameter, scale, is a scalar.

`CInvDisKernel`

`Kappa(x)` Given a numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \|\mathrm{x1_i} - \mathrm{x2_j}\|^2$, it computes the kernel of one over the distance of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \frac{1}{\sqrt{\mathrm{x}}} \tag{5.6}$$

`CInvSqDisKernel`

`Kappa(x)` Given a numpy.array, x with entries $(\mathrm{x})_{\mathrm{ij}} = \|\mathrm{x1_i} - \mathrm{x2_j}\|^2$, it computes the kernel of one over the square of distance of the form:

$$\mathrm{Kappa}(\mathrm{x}) = \frac{1}{\mathrm{x}} \tag{5.7}$$

Note that will call the common parts of the kernels as the base part. Each kernel is defined with respect to these common parts by Kappa function. For instance, all dot product kernels have base part $\langle \mathrm{x1_i}, \mathrm{x2_j} \rangle$, and all RBF kernels have base part $\|\mathrm{x1_i} - \mathrm{x2_j}\|^2$. For example, the polynomial kernel is simply $\mathrm{Kappa}(\langle \mathrm{x1_i}, \mathrm{x2_j} \rangle) = (\mathrm{scale} \cdot \langle \mathrm{x1_i}, \mathrm{x2_j} \rangle + \mathrm{offset})^{\mathrm{degree}}$ and the Gauss kernel is simply $\mathrm{Kappa}(\|\mathrm{x1_i} - \mathrm{x2_j}\|^2) = \exp(-\mathrm{scale} \cdot \|\mathrm{x1_i} - \mathrm{x2_j}\|^2)$. Basically, in the implementations, efficient computations focus mainly on the base part. The specialization into different kernels are done by implementing different Kappa functions.

5.5.3 Calling Examples

In this subsection, we will give a piece of example codes. These codes illustrate the major modes of using the methods of the vector kernels. Note that except for the arguments x1, x2, y1, y2 and alpha that can be passed into the methods,

there are three optional arguments that can be used. These arguments are index1, index2, and output. The first two optional arguments allows kernel matrix manipulation on a subset of the data. The third optional argument allows the users explicitly provide a buffer and thus can be more memory efficient.

```
import numpy
import numpy.random as random

datano =10
dimno = 2

# Generate the data
x1 = random.randn((datano,dimno))
x2 = random.randn((datano, dimno))

# Generate the labels
y1 = numpy.array([[1,1,1,1,1,-1,-1,-1,-1,-1]])
y1.shape = (datano, )
y2 = numpy.array([[1,1,1,1,1,-1,-1,-1,-1,-1]])
y2.shape = (datano, )

# Generate the vector of coefficients
alpha = random.randn((datano, 1))

# Indices to the samples
idx1 = [0, 1, 5, 8]
idx2 = [9]

# Create kernel object
scale = 0.1
kernel = CGaussKernel(scale)

# Create output buffer
buffer = numpy.zeros((datano, datano))

# Compute the full kernel matrix for data set x1 and x2
k1 = kernel.Dot(x1, x2)

# Compute the kernel matrix only using the data in x1 indexed by
idx1
k2 = kernel.Dot(x1, x2, idx1)


# Compute the kernel matrix only using the data in x2 indexed by
idx2
k3 = kernel.Dot(x1, x2, index2=idx2)

# Compute the full kernel matrix and store it in buffer
k4 = kernel.Dot(x1, x2, output=buffer)

# Compute the kernel matrix only using the data in x1 indexed by
idx1, and store it in buffer
k5 = kernel.Dot(x1, x2, index1=idx1, output=buffer)

# Compute the kernel matrix only using the data in x2 indexed by #
idx2, and store it in buffer
k6 = kernel.Dot(x1, x2, index2=idx2, output=buffer)

# Compute the kernel matrix only using the data in x1 indexed by #
idx1 and the data in x2 indexed by idx2, and store it in buffer
k7 = kernel.Dot(x1, x2, idx1, idx2, buffer)

# Create another output buffer
buffer = numpy.zeros((datano, 1))
```



```
# Compute the tensor expansion vector using data sets x1 and x2,
labels y1 and y2, and the coefficients alpha
v1 = kernel.TensorExpand(x1, y1, x2, y2, alpha)

# Compute the tensor expansion vector using data sets x1 and x2
```

5.5.4 Additional functionalities

Several extra functionalities are added to the vector kernels to allow decremental update of the kernel matrices. This set of functions basically cache the full kernel matrix and compute new kernel matrix by deleting one or several dimensions of the data at a time. These functionalities facilitate backward elimination for feature selection using these kernels. The interface of these functions are:

`CreateCacheKernel(x1)` Cache the

base part of the kernel matrix of the data set x1.

`ClearCacheKernel(x1)` Clear the

cache for the base part of the kernel matrix of the data set x1.

`DotCacheKernel(x1)` Compute

the kernel matrix of the data set x1 base on the cached base part by applying Kappa functions.

`DecCacheKernel(x1, x2)` Decrement

the base part of the kernel matrix of the data set x1 by the base part of the kernel matrix of the data set x2. In this case, x2 is usually taken from several dimensions of x1. This operation changes the content in the cache.

`DecDotCacheKernel(x1, x2)` Temporally

decrement the base part of the kernel matrix of the data set x1 by the base part of the kernel matrix of the data set x2, and then return the resulting kernel matrix. In this case, x2 is usually taken from several dimensions of x1. This operation dos *not* change the content in the cache.

`GradDotCacheKernel(x1)` Compute

the gradient of each entry in the kernel matrix of the data set x1 with respect to the kernel parameters. Note that current this operation only supports the gradient with respect to a single scalar parameter.

`KappaGrad(x)` Compute

the gradient of the kernel in term of the base part x (numpy.array) with respect to a kernel parameter. This function is provided in the same vain as the Kappa function. Overload this method to compute gradients differently for different kernels.

`SetParam(param)` Set the pa-

rameters for the kernel. Note that the format of the parameter is not specified in this common interface. Different kernel may interpret param differently. For instance, two RBF kernel CGaussKernel and CLaplaceKernel use interpret param as numpy.array and use param[0] to set the scale parameter.

For more information on how to use this function, an example is provided in the testing functions for this package.

5.6 Kernels on structured data

5.6.1 Installation Guide

1. Install Boost.Python
2. Issue a `bjam` command under the directory elefant/kernels/stringkernel/src
3. Once the shared libraries `SK.so` and `libboost_python.so.versionnumber` are built, copy them (from directories under elefant/kernels/stringkernel/src/bin) into the directory elefant/kernels/stringkernel/bin
4. Include those shared libraries in the environment variable LD_LIBRARY_PATH

5.6.2 String Kernels

The class CStringKernel operates on strings (hence the name structured data). The string kernel is defined as the inner product of feature vectors of the given strings. Each non-zero entry (or coordinate) of these feature vectors indicates the occurrence of a particular substring in the strings. The efficient computation of the kernel above can be formulated as,

$$K(x, x') = \sum_{s \in \mathcal{A}^*} num_s(x) \cdot num_s(x') \cdot w_s$$

where x and x' are two strings; s a string in the set of all possible strings $\mathcal{A}^*$; $num_s(x)$ is the number of times substring s occurs in x and w_s is the weight associated to s.

Essentially, this all-substrings kernel is computed by mean of string matching between two given strings. Hence, the all-substring kernel and its variants can be realised through the use of different substrings weighting functions (SWF), i.e. w_s. The following is the brief description of the implemented string kernel variants (see Teo and Vishwanathan (2006); Vishwanathan and Smola (2004) for details).

Constant (`constant`) : The kernel considers all matching substrings and assigns constant weight (e.g. 1) to each of them. This SWF does not require any parameter.

Exponential Decay (`expdecay`) : The substring weight decays as the matching substring gets longer. The SWF require a decay factor λ in the range (1,∞).

k-Spectrum (`kspectrum`) : The kernel considers only matching substring of exactly length k. Each such matching substring is given a constant weight. This SWF require the parameter k in the range [1,∞).

Bounded range (`boundedrange`) : The kernel considers only matching substring of length less than or equal to a given number N. This SWF requires parameter N in the range [1,∞)[1].

5.6.3 CStringKernel Methods

Since CStringKernel is a subclass of CKernel, it shares common method interfaces with other kernels. However, the output (i.e. kernel values) of all CStringKernel methods are normalised except the method $K(x1, x2)$. A normalised version is provided as $NormalisedK(x1, x2)$. Besides, CStringKernel requires users to provide the SWF type and its corresponding parameter when instantiating a particular CStringKernel object.

`CStringKernel(swf="constant", swfParam="0.0")`
The constructor for class CStringKernel.

[1] **Bag-of-Characters** kernel is a special case of Bounded range kernel when $N = 1$.

swf Type of substring weighting function (swf). This argument is passed as a Python string object. Possible choices of swf are `constant`, `expdecay`, `kspectrum`, and `boundedrange`. See section 5.6.2 for details.

swfParam Parameter for chosen swf. Note that, `constant` does not require any parameter.

Examples:
`myConstSK = CStringKernel()` `myKSpecSK = CStringKernel("kspectrum", 5)`

`K(x1, x2)`
See section 5.4 for description.

x1 Data point of type Python string. CStringKernel build an ESA **?** for **x1** in order to compute kernel value.

x2 Data point of type Python string.

`NormalisedK(x1, x2)`
Similar to `K(x1, x2)` but output is normalised by a multiplicative factor equals to $1/\sqrt{K(x1, x1) \cdot K(x2, x2)}$.

`Dot(x1, x2, index1=None, index2=None, output=None)`
See section 5.4 for description.

x1 Data points as a List of Python string.

x2 Data points as a List of Python string.

index1 Numpy 1-D array (or Python list) of integers.

index2 Numpy 1-D array (or Python list) of integers.

output Numpy 2-D array (or Python list) of real numbers.

`DotKM(x1, index1=None, output=None)`
Special case of Dot() when **x1** and **index1** are identical to **x2** and **index2**, respectively.

`Expand(x1, x2, alpha2, index1=None, index2=None, output=None)`
See section 5.4 for description.

x1 Data points as a List of Python string.

x2 Data points as a List of Python string

alpha2 Numpy 1-D array (or Python list) of real numbers.

index1 Numpy 1-D array (or Python list) of integers.

index2 Numpy 1-D array (or Python list) of integers.

output Numpy 2-D array (or Python list) of real numbers.

`Tensor(x1, y1, x2, y2, index1=None, index2=None, output=None)`
See section 5.4 for description.

x1 Data points as a List of Python string.

y1 Numpy 1-D array (or Python list) of real numbers.

x2 Data points as a List of Python string

y2 Numpy 1-D array (or Python list) of real numbers.

index1 Numpy 1-D array (or Python list) of integers.

index2 Numpy 1-D array (or Python list) of integers.

output Numpy 2-D array (or Python list) of real numbers.

`TensorKM(x1, y1, index1=None, output=None)`
Special case of Tensor() when **x1**, **y1** and **index1** are identical to **x2**, **y2** and **index2**, respectively.

`TensorExpand(x1, y1, x2, y2, alpha2, index1=None, index2=None, output=None)`
See section 5.4 for description.

x1 Data points as a List of Python string.

y1 Numpy 1-D array (or Python list) of real numbers.

x2 Data points as a List of Python string

y2 Numpy 1-D array (or Python list) of real numbers.

alpha2 Numpy 1-D array (or Python list) of real numbers.

index1 Numpy 1-D array (or Python list) of integers.

index2 Numpy 1-D array (or Python list) of integers.

output Numpy 2-D array (or Python list) of real numbers.

Once the CStringKernel object is instantiated, it can be used by any kernel methods.

5.6.4 Implementation Notes

- The data accepted by CStringKernel is of type Python string. Hence, string dataset should be stored as a Python list of strings.
- The (currently implemented) underlying data structure of CStringKernel (i.e. Enhanced Suffix Arrays (ESA) (see Abouelhoda et al. (2004))) and the fast (kernel) matrix vector multiplication append a SENTINEL character '\n' to each string data point. Hence, the character '\n' is assumed to be absent throughout the dataset.
- The memory requirement of CStringKernel is around 19 to 21 bytes per input character. The variation in memory requirement depends on the quality of compression of Longest Common Prefix (LCP) table in ESA, which in turn depends on the given strings. Highly repetitive strings are more likely to increase the memory usage of LCP table.

BIBLIOGRAPHY

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. 2:53–86, 2004.

Bernard Schölkopf and Alexander J. Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. 2001.

C. H. Teo and S. V. N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 929–936, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-383-2. doi: http://doi.acm.org/10.1145/1143844.1143961.

S. V. N. Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. In K. Tsuda, B. Schölkopf, and J.P. Vert, editors, *Kernels and Bioinformatics*, Cambridge, MA, 2004.

CHAPTER

SIX

Kernel Hebbian Algorithm

6.1 Overview

This package provides the Kernel Hebbian Algorithm (KHA) introduced in Kim et al. (2005) and its improved version introduced in Schraudolph et al. (2007). The Kernel Hebbian Algorithm iteratively approximates the result of a Kernel Principal Component Analysis.

6.2 Short Description

Principal Components Analysis (PCA) is a standard linear technique for dimensionality reduction. Given a matrix $\mathbf{X} \in \Re^{n \times l}$ of l centered, n-dimensional observations, PCA performs an eigendecomposition of the covariance matrix $\mathbf{Q} := \mathbf{X}\mathbf{X}^\top$. The $r \times n$ matrix $\mathbf{W}$ whose rows are the eigenvectors of $\mathbf{Q}$ associated with the $r \leq n$ largest eigenvalues minimizes the least-squares reconstruction error

$$\|\mathbf{X} - \mathbf{W}^\top \mathbf{W}\mathbf{X}\|_F, \tag{6.1}$$

where $\|\cdot\|_F$ is the Frobenius norm.

As it takes $O(n^2 l)$ time to compute $\mathbf{Q}$ and $O(n^3)$ time to eigendecompose it, PCA can be prohibitively expensive for large amounts of high-dimensional data.

Iterative methods exist that do not compute $\mathbf{Q}$ explicitly, and thereby reduce the computational cost to $O(rn)$ per iteration. They assume that each individual observation $\vec{x}$ is drawn from a statistical distribution[1], and the aim is to maximize the variance of $\vec{y} := \mathbf{W}\vec{x}$, subject to some orthonormality constraints on the weight matrix $\mathbf{W}$. Such a method is the *Generalized Hebbian Algorithm* (GHA) of Sanger (1989).

Kim et al. (2005) adapt Sanger's (1989) GHA algorithm to work with data mapped into a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ via a feature map $\Phi : \mathcal{X} \to \mathcal{H}$ which resulted in the Kernel Hebbian Algorithm (KHA).

In Schraudolph et al. (2007) improvements to KHA were presented. KHA is accelerated by incorporating the reciprocal of the current estimated eigenvalues as part of a gain vector and by applying Stochastic Meta-Descent (SMD) gain vector adaptation (Schraudolph, 2002, 1999) in reproducing kernel Hilbert space.

6.3 Component Interface

The KHA component has four ports:

- **data**: Data set stored as numpy DenseMatrix where the i-th row contains the values of the i-th element. This port must be connected before executing the algorithm.

[1] It is customary to assume that the distribution is centered, i.e. $E[\vec{x}] = \vec{0}$.

- **A**: Output port for the coefficient matrix $\mathbf{A}$ as numpy DenseMatrix. The i-th row of $\mathbf{A}$ contains the coefficients of the i-th eigenvector.
- **AK**: Output port for the product of $\mathbf{A}$ with the kernel matrix as numpy DenseMatrix.
- **error**: Output port for the numpy DenseMatrix containing the reconstruction errors. The value error[i][0] is the reconstruction error after the (i+1)-th pass through the data set.

6.4 Algorithm parameters

In order to execute the Kernel Hebbian algorithm the following steps must be done:

1. Generate object of class CKHA.
2. Connect all ports of the object.
3. Execute KHA method of the object.

The KHA method has a large number of mandatory and optional parameters:

- *repnum:* Number of passes through whole data set the KHA will execute.
- *rows:* Number of eigenvectors KHA will estimate.
- *eigen_type:* Type of algorithm used:
  - 0: Standard KHA, i.e. no eigenvalues used or estimated.
  - 1: Standard KHA but matrix AK is maintained allowing the calculation of reconstruction error.
  - 2: KHA/et*, learning rate of eigenvectors are multiplied by 1/ estimated eigenvalue.
  - 3: KHA/et learning rate of eigenvectors are multiplied by 1/ estimated eigenvalue and length of the vector of eigenvalues. For the first 200 iterations no eigenvalue scaling is used (grace period).

  If we use $mu > 0$ the eigen_type 2 and 3 will lead to KHA/et-SMD* and KHA-SMD, respectively.
- *estart:* learning rate for first KHA step, corresponds to η_0.
- *tau:* learning rate decay factor, corresponds to τ; base learning rate is $\frac{\eta_0 \cdot \tau}{\tau + \mathrm{it}}$ where it is the number of learning rate updates. In the special case of $\tau = 0$ the base learning rate is $\frac{\eta_0}{1+\mathrm{it}}$.
- *mu:* SMD meta-stepsize parameter μ. If 0 then SMD is disabled.
- *lambd:* SMD trace history decay parameter λ. Value 1 corresponds to no decay, 0 to only using last step's information.
- *sigma:* σ parameter for the Gaussian Kernel. This is only used if parameter *kernel* is None. Default value is 1.
- *loadtemp:* The KHA method stores some information in files AK and krsum to allow quick initialization of the algorithm if used with the same data set. If the value for this parameter is 1 then the initialization information is loaded. This should only be done if the algorithm was started with the same data set, parameters *sigma* and *kernel* before. Default value is 0.
- *filenamebase:* The initial part of the name of the files where the results are saved. The saved results are *petsc* matrices A, AK and the reconstruction errors. If the value of this parameter is None (default value) then no results are saved.
- *verbose:* If the value of this parameter is 1 then temporary results are saved all 100 KHA iterations instead of all 10000 KHA iterations. In addition in case of value 1 the iteration number is output after each iteration. Default value is 0.

- *kernel:* Kernel of type elefant.kernels.vector used for computing Kernel matrix. If None (default) then a Gaussian kernel implemented in C++ will be used with indicated value of σ.
- *update_frequency* Frequency f of updating learning rate and eigenvalue estimates. The value it mentioned in description of parameter *tau* is the number of KHA iterations divided by f. Default value is 1, i.e. there is an update in each iteration.

6.5 Examples and Unit Test

The following examples and unit test programs are provided:

- *example1.py:* KHA of 1000 USPS digits using both kernels implemented in C++ and python. 16 KPCAs are searched for.
- *example2.py:* KHA of a part of the Lena image where the image is divided in large number of 11×11 patches which are regarded as the input patters. 20 KPCAs are searched for.
- *example3.py:* KHA of 1000 MNIST digits. 20 KPCAs are searched for.
- *example4.py:* KHA of the MNIST data set with five input parameters:

  1. Number of digits used (maximum 60000).
  2. Type of algorithm (corresponds to *eigen_type* parameter of KHA).
  3. Start learning rate η_0.
  4. Decay parameter; will be multiplied by first parameter to get τ used by KHA.
  5. Value of parameter μ.

  50 KPCAs are searched for.
- *utest.py:* KHA of 1000 USPS digits using both kernels implemented in C++ and python and comparison of their results and run time. 16 KPCAs are searched for. This example shows how the KHA module can be used in a component framework.

BIBLIOGRAPHY

K. I. Kim, M. O. Franz, and B. Schölkopf. Iterative kernel principal component analysis for image modeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(9):1351–1366, 2005.

T. D. Sanger. Optimal unsupervised learning in a single-layer linear feedforward network. *Neural Networks*, 2: 459–473, 1989.

Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

Nicol N. Schraudolph. Local gain adaptation in stochastic gradient descent. pages 569–574, Edinburgh, Scotland, 1999. IEE, London.

Nicol N. Schraudolph, Simon Günter, and S.V.N. Vishwanathan. Fast iterative kernel pca. In *Advances in Neural Information Processing Systems*, volume 19. MIT Press, 2007.

CHAPTER

SEVEN

Optimization

7.1 Introduction

This chapter describes a set of optimization methods. Some are specific to ELEFANT, while others are adaptations of existing packages, such as SciPy to suit the problems occurring in machine learning. All of them could be used in a larger context.

At present, the only documented module is a quadratic programming code based on a Primal-Dual Interior Point method suggested in Vanderbei (1992). A detailed description is given in Section **??**. It is a section taken from Schölkopf and Smola (2002).

7.1.1 Known Issues

- The solver currently does not handle infinities explicitly. Instead, it always assumes that the bound constraints are given. This needs to be implemented. Basically, what needs to be done is have a case-by-case treatment when variables are missing. And obviously, such a treatment should not require for-loops but instead be using vectorization.
- The solver currently does not scale well beyond problems which can be fully stored in main memory. This is so, as it requires a matrix factorization to solve the optimization problem. A subset-selection and chunking method would fix this.
- The solver currently is not able to perform hot starts. This is a general problem with interior point methods. Gondzio's work would have pointers on how to fix it.
- Dahl and Vandenberghe's CVXOPT is probably orders of magnitude better and more flexible than what is here. That said, at the moment their license (GPL) and the inclusion of certain packages (UMFPACK) makes their code unsuitable for commercially relevant use in our case.

7.1.2 File Hierarchy

The files used in this document are all contained in the *optimization* directory of Elefant. The hierarchy of the files contained therein is as follows:

core:

intpointsolver.py	The Interior Point Solver Core
quadpart.py	This part solves the reduced KKT System
linesearch.py	Performs line search (from SciPy)
matrix_linesearch.py	Performs line search (for matrices, from SciPy)
newton_CG.py	Newton-Conjugate Gradient solver (from SciPy)
newton_CG.py	Newton-Conjugate Gradient solver (for matrices, from SciPy)

utest:

utest/utest.py	Unit testing framework

7.2 Quick-Start Guide

This guide should be sufficient in order to begin using the algorithms as quickly as possible. We'll begin by just looking at the classes that the user needs, and some simple code segments. If you're feeling confident, you might skip this section altogether, and proceed straight to some of the simpler examples – these are covered in section **??**.

The two main solvers are a Linear Programming and a Quadratic Programming Solver. In order to solve the Linear Program

$$\text{minimize } c^\top x \text{ subject to } b \leq Ax \leq b + d \text{ and } l \leq x \leq u \tag{7.1}$$

you need to do the following

```
mysolver = intpointsolver.CLPSolver()
x, y, how = mysolver.Solve( c, A, b, r, l, u)
```

Here `x` will be the solution of the problem, `y` contains the dual variables, and `how` describes whether the optimization problem converged. The first line creates an interior point solver object. The second line calls the solver to solve a specific instance, as specified by the parameters.

Likewise, in order to solve the Quadratic Program

$$\text{minimize } \frac{1}{2}x^\top Hx + c^\top x \text{ subject to } b \leq Ax \leq b + d \text{ and } l \leq x \leq u \tag{7.2}$$

the following

```
mysolver = intpointsolver.CLPSolver()
x, y, how = mysolver.Solve( c, H, A, b, r, l, u)
```

7.3 API Reference

7.3.1 Interior Point Solvers

The interior point solvers are made up of two modules, **intpointsolver** and **quadpart**. The former contains the core of the interior point solver algorithm. It provides the following classes:

CIntpointSolver
This defines the general interior point solver object which deals with the general structure of an interior point solver. Methodical changes need to occur here. All classes below are derived from it.

CIntpointSolver::__init__(maxsigfig=7, maxiter=50, margin=0.05, verbose=0, bound=10)
This is the constructor for the generic solver class. It only sets parameters relevant for the convergence and termination of the solver itself.

Input Parameters

maxsigfig: An integer specifying the number of significant digits within which optimality needs to be achieved before convergence is assumed. Setting it to 7 is considerably more conservative than what most applications in machine learning require but it makes good sense for a general purpose optimization algorithm.
maxiter: Maximum number of iterations. The algorithm should rarely take more than 20 iterations. So 50 is a good upper bound for when things start going wrong.
margin: The proximity to the boundary for the next update step. Moving too closely may result in bad convergence. Note that this is utterly unrelated to the margin in SVMs.
verbose: Changes the level of verbosity. We have 0 - quiet, 1 - only report convergence, 2 - report progress at every iteration. Larger values than that are equivalent to setting it to 2.
bound: Determines the boundary for the initialization. 10 is typically a good value. Probably a smarter problem-dependent setting would be useful.

CIntpointSolver::GenericSolve(c, hx, a, b, r, l, u)
This describes a generic solver method for convex constrained optimization problems. The key part in it is hx, which contains the quadratic part of the optimization problem and solves the reduced Karush-Kuhn-Tucker problem. Its various instances can be found in the module **quadpart**. See the description there.

Input Parameters

c: Linear term of dimension m.
hx: Object containing information on how to solve the reduced KKT system.
a: Constraint matrix of dimension $n \times m$. This is part of the constraint

$$b \leq Ax \leq b + r \tag{7.3}$$

b: Lower bound on the constraint system of dimension n.
r: Increment of the constraint system. Again of dimension n.
l: Lower bound on x. Dimensionality is m.
u: Upper bound on x. Dimensionality is m.

Output Parameters

x: The solution of the optimization problem.
y: The dual variables. In SVMs this is often equivalent to the constant offset b.
how: Possible outcomes are ('converged', 'primal and dual infeasible', 'primal infeasible', 'dual infeasible', and 'slow convergence, change bound?'). The result is a string.

CQPSolver
Derived from CIntpointSolver. This solves the quadratic program described in (7.2).

CQPSolver::Solve(c, hx, a, b, r, l, u)
It behaves in a manner identical to **CIntpointSolver::GenericSolve**. The only difference is hx.

Input Parameters

hx: Positive Semidefinite quadratic matrix of size $m \times m$. Note that the solver does not check whether hx satisfies this condition. If it does not, the optimization may fail.

CLPSolver
CLPSolver::Solve(c, a, b, r, l, u)
It behaves in a manner identical to **CIntpointSolver::GenericSolve**. However, it does not contain a quadratic part, as it is a purely linear program.

CRegressionSolver
This solves the quadratic program described in (7.2). However, to deal with the special structure of a regression optimization problem, it takes advantage of the fact that the quadratic matrix H is given by

$$H = \begin{bmatrix} K + D_1 & -K \\ -K & K + D_2 \end{bmatrix} \tag{7.4}$$

CRegressionSolver::Solve(c, hx, d1, d2, a, b, r, l, u)
Solves the type of quadratic programs common in regression estimation.

Input Parameters

hx: Positive Semidefinite quadratic matrix of size $\frac{m}{2} \times \frac{m}{2}$, which corresponds to K in (7.4).
d1: Diagonal matrix with nonnegative entries, stored as a vector of size $\frac{m}{2}$ (this corresponds to D_1). In SVM Regression these entries will all be zero. For Huber's robust regression they are all nonnegative and constant.
d2: Diagonal matrix with nonnegative entries, stored as a vector of size $\frac{m}{2}$ (this corresponds to D_2).

CSMWSolver
This solves the quadratic program described in (7.2). However, it takes advantage of the structure inherent in low-rank problems. Here H is given by

$$H = ZZ^\top \tag{7.5}$$

CSMWSolver::Solve(c, z, a, b, r, l, u)
It behaves in a manner identical to **CQPSolver:GenericSolve**. The only difference is z, which describes the factors in the quadratic problem. The Sherman-Morrison-Woodbury formula is used. Note: instead of the SMW decomposition one should rather use the one described in Vishwanathan (2002), which is essentially a fancy $LDL^\top$ decomposition.

Input Parameters

z: The design matrix, as given by (7.5), is an $m \times d$ object, where d is the number of dimensions.

CSMWRegressionSolver
This is a combination of the features in `CSMWSolver` and `CRegressionSolver`. This means that the quadratic part of the system is made up of (7.4), where the dense quadratic system is determined by (7.5). Consequently the call signature looks as follows.

CSMWRegressionSolver::Solve(c, z, d1, d2, a, b, r, l, u)
The arguments follow the same conventions as in the SMW and the Regression solvers.

7.3.2 Reduced Karush Kuhn Tucker Systems

When trying to expand the interior point codes, one may want to modify the module `quadpart`, as it contains code to solve the following reduced KKT system:

$$\begin{bmatrix} -H - D & A^\top \\ A & E \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c_x \\ c_y \end{bmatrix} \tag{7.6}$$

Note: this code is not meant for a casual user of the solver. Instead, its use is meant for users wishing to expand the solver. The code itself contains ample documentation. Below we give an overview over the methods one needs to implement to adapt the solver to one's needs.

CGeneric
Generic KKT Solver class. The constructor initializes the quadratic terms.

CGeneric::__init__(hx)
Sets the quadratic term of the KKT system. For fancy decompositions use a more sophisticated constructor instead. The methods **VecMult**, **PresolveKKT**, and **SolveKKT** are the only ones exposed to the solver code in the interior point solver.

CGeneric::SetDiag(v, m=None)
Sets diagonal entries in a matrix m. If no matrix is provided, it creates a dense matrix with v as diagonal entries.

This is an internal helper routine. No need to re-implement this for any derived class.

CGeneric::VecMult(v)
Performs matrix-vector multiplication. It returns Hv, where H is the internally stored quadratic matrix.

CGeneric::PresolveKKT(a, d, e)
Presolves the reduced KKT system (7.6), whenever possible. Here a is the constraint matrix, d the diagonal term, and e the term corresponding to a quadratic regularization in the constraints.

Note: you must call this before calling **SolveKKT**, whenever a, d, e change.

CGeneric::SolveKKT(cx, cy)
Computes the solution of the reduced KKT system. It returns **x, y**, the solution of the problem.

CEmpty
This is used for the linear programming solver. Here the quadratic part is empty.

CRegression
This uses the matrix decomposition as it is typical in SVM regression.

CSMW
Sherman Morrison Woodbury decomposition, as described in Schölkopf and Smola (2002).

CSMWRegression
Sherman Morrison Woodbury decomposition for regression.

7.4 Scientific Background

7.4.1 Interior Point Methods

We describe the inner workings of the interior point solver based on the idea of solving the following quadratic problem:

$$\text{minimize } \frac{1}{2}x^\top Hx + c^\top x \text{ subject to } Ax + d \leq 0 \text{ and } l \leq x \leq u. \tag{7.7}$$

That is, the problem of (7.2). Interior point solvers Nesterov and Nemirovskii (1993); Vanderbei (1992) work by enforcing primal and dual feasibility of the corresponding optimization problems.

We modify the constraints slightly. Rather than using the inequality constraint $Ax + d \leq 0$ directly, we are better off with an equality and a positivity constraint for an additional variable, i.e. we transform $Ax+d \leq 0$ into $Ax+d+\xi = 0$, where $\xi \geq 0$. Hence we arrive at the following system of equations:

$$\begin{array}{lcll} Kx + A^\top\alpha + c & = & 0 & \text{(Dual Feasibility),} \\ Ax + d + \xi & = & 0 & \text{(Primal Feasibility),} \\ \alpha^\top\xi & = & 0, & \\ \alpha, \xi & \geq & 0. & \end{array} \tag{7.8}$$

Interior point methods work by satisfying the linear constraints of the above set, while ensuring that the KKT conditions are satisfied, i.e. that $\alpha^\top\xi = 0$ holds. The solver starts by finding a point which is primal and dual approximately feasible (i.e. which satisfies the linear constraints) and then subsequently enforces the KKT conditions.

Let us analyze the equations in more detail. We have three sets of variables: x, α, ξ. To determine the latter, we have an equal number of equations plus the positivity constraints on α, ξ. While the first two equations are linear and thus amenable to solution, e.g., by matrix inversion, the third equality $\alpha^\top\xi = 0$ has a small defect: given one variable, say α, we cannot solve it for ξ or vice versa. Furthermore, the last two constraints are not very informative either.

We use a primal-dual path-following algorithm, as proposed in Vanderbei (1997), to solve this problem. Rather than requiring $\alpha^\top\xi = 0$ we modify it to become $\alpha_i\xi_i = \mu > 0$ for all $i \in [n]$, solve (7.8) for a given μ, and decrease μ to 0 as we go. The advantage of this strategy is that we may use a Newton-type predictor corrector algorithm to update the parameters x, α, ξ, which exhibits the fast convergence of a second order method.

Solving the Equations For the moment, assume that we have suitable initial values of x, α, ξ, and μ with $\alpha, \xi > 0$. Linearization of the first three equations of (7.8), together with the modification $\alpha_i\xi_i = \mu$, yields (we expand x into $x + \Delta x$, etc.):

$$\begin{array}{lcll} K\Delta x + A^\top\Delta\alpha & = & -Kx - A^\top\alpha - c & =: \rho_p, \\ A\Delta x + \Delta\xi & = & -Ax - d - \xi & =: \rho_d, \\ \alpha_i^{-1}\xi_i\Delta\alpha_i + \Delta\xi_i & = & \mu\alpha_i^{-1} - \xi_i - \alpha_i^{-1}\Delta\alpha_i\Delta\xi_i & =: \rho_{\mathrm{KKT}\,i} \text{ for all } i \end{array} \tag{7.9}$$

Next we solve for $\Delta\xi_i$ to obtain what is commonly referred to as the *reduced* KKT system. For convenience we use $D := \operatorname{diag}(\alpha_1^{-1}\xi_1, \ldots, \alpha_n^{-1}\xi_n)$ as a shorthand;

$$\begin{bmatrix} K & A^\top \\ A & -D \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\alpha \end{bmatrix} = \begin{bmatrix} \rho_p \\ \rho_d - \rho_{\mathrm{KKT}} \end{bmatrix}. \tag{7.10}$$

We apply a predictor-corrector method. The resulting matrix of the linear system in (7.10) is indefinite but of full rank, and we can solve (7.10) for $(\Delta x_{\mathrm{Pred}}, \Delta\alpha_{\mathrm{Pred}})$ by explicitly pivoting for individual entries (for instance, solve for Δx first and then substitute the result in to the second equality to obtain $\Delta\alpha$).

This gives us the *predictor* part of the solution. Next we have to correct for the linearization, which is conveniently achieved by updating ρ_{KKT} and solving (7.10) again to obtain the *corrector* values $(\Delta x_{\mathrm{Corr}}, \Delta\alpha_{\mathrm{Corr}})$. The value of $\Delta\xi$ is then obtained from (7.9).

Next, we have to make sure that the updates in α, ξ do not cause the estimates to violate their positivity constraints. This is done by shrinking the length of $(\Delta x, \Delta\alpha, \Delta\xi)$ by some factor $\lambda \geq 0$, such that

$$\min\left(\frac{\alpha_1 + \lambda\Delta\alpha_1}{\alpha_1}, \ldots, \frac{\alpha_n + \lambda\Delta\alpha_n}{\alpha_n}, \frac{\xi_1 + \lambda\Delta\xi_1}{\xi_1}, \ldots, \frac{\xi_n + \lambda\Delta\xi_n}{\xi_n}\right) \geq \epsilon. \tag{7.11}$$

Of course, only the negative Δ terms pose a problem, since they lead the parameter values closer to 0, which may lead them into conflict with the positivity constraints. Typically Smola (1998); Vanderbei (1997), we choose $\epsilon = 0.05$. In other words, the solution will not approach the boundaries in α, ξ by more than 95%.

Updating μ Next we have to update μ. Here we face the following dilemma: if we decrease μ too quickly, we will get bad convergence of our second order method, since the solution to the problem (which depends on the value of μ) moves too quickly away from our current set of parameters (x, α, ξ). On the other hand, we do not want to spend too much time solving an *approximation* of the unrelaxed ($\mu = 0$) KKT conditions *exactly*. A good indication is how much the positivity constraints would be violated by the current update. Vanderbei Vanderbei (1997) proposes the following update of μ:

$$\mu = \frac{\alpha^\top \xi}{n}\left(\frac{1-\lambda}{10+\lambda}\right)^2. \tag{7.12}$$

The first term gives the average value of satisfaction of the condition $\alpha_i\xi_i = \mu$ after an update step. The second term allows us to decrease μ rapidly if good progress was made (small $(1-\lambda)^2$). Experimental evidence shows that it pays to be slightly more conservative, and to use the *predictor* estimates of α, ξ for (7.12) rather than the corresponding corrector terms. This imposes little overhead for the implementation.

Initial Conditions and Stopping Criterion To provide a complete algorithm, we have to consider two more things: a stopping criterion and a suitable start value. For the latter, we simply solve a regularized version of the initial reduced KKT system (7.10). This means that we replace K by $K + \text{one}$, use (x, α) in place of $\Delta x, \Delta\alpha$, and replace D by the identity matrix. Moreover, ρ_p and ρ_d are set to the values they would have if all variables had been set to 0 before, and finally ρ_{KKT} is set to 0. In other words, we obtain an initial guess of (x, α, ξ) by solving

$$\begin{bmatrix} K + \text{one} & A^\top \\ A & -\text{one} \end{bmatrix}\begin{bmatrix} x \\ \alpha \end{bmatrix} = \begin{bmatrix} -c \\ -d \end{bmatrix}, \tag{7.13}$$

and $\xi = -Ax - d$. Since we have to ensure positivity of α, ξ, we simply replace

$$\alpha_i = \max(\alpha_i, 1) \text{ and } \xi_i = \max(\xi_i, 1). \tag{7.14}$$

This heuristic solves the problem of a suitable initial condition.

To obtain a stopping criterion we need to ensure that $f(x)$ is close to its optimal value $f(\bar{x})$. We know, provided the feasibility constraints are all satisfied, that the value of the dual objective function is given by $f(x) + \sum_{i=1}^n \alpha_i c_i(x)$. We may use the latter to bound the *relative* size of the gap between primal and dual objective function by

$$\text{Gap}(x, \alpha) = \frac{2\left|f(x) - \left(f(x) + \sum_{i=1}^n \alpha_i c_i(x)\right)\right|}{|f(x)| + \left|\left(f(x) + \sum_{i=1}^n \alpha_i c_i(x)\right)\right|} \leq \frac{-\sum_{i=1}^n \alpha_i c_i(x)}{\left|f(x) + \frac{1}{2}\sum_{i=1}^n \alpha_i c_i(x)\right|}. \tag{7.15}$$

For the special case where $f(x) = \frac{1}{2}x^\top Kx + c^\top x$ we know that the size of the feasibility gap is given by $\alpha^\top\xi$, and therefore

$$\text{Gap}(x, \alpha) = \frac{\alpha^\top \xi}{\left|\frac{1}{2}x^\top Kx + c^\top x + \frac{1}{2}\alpha^\top\xi\right|}. \tag{7.16}$$

In practice, a small number is usually added to the denominator of (7.16) in order to avoid divisions by 0 in the first iteration. The quality of the solution is typically measured on a logarithmic scale by $-\log_{10}\text{Gap}(x, \alpha)$, the number of significant figures.

Primal-Dual path following methods are certainly not the only algorithms that can be employed for minimizing constrained quadratic problems. Other variants, for instance, are Barrier Methods Bertsekas (1995); Karmarkar (1984); Vanderbei et al. (1994), which minimize the unconstrained problem

$$f(x) + \mu \sum_{i=1}^{n} f \ln \left(-c_i(x)\right) \text{ for } \mu > 0. \tag{7.17}$$

Active set methods have also been used with success in machine learning Kaufman (1999); More and Toraldo (1991). These select subsets of variables x for which the constraints c_i are not active, i.e., where the we have a strict inequality, and solve the resulting restricted quadratic program, for instance by conjugate gradient descent.

BIBLIOGRAPHY

D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 1995.

N. K. Karmarkar. A new polynomial-time algorithm for linear programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

L. Kaufman. Solving the quadratic programming problem arising in support vector classification. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods - -Support Vector Learning*, pages 147–168, Cambridge, MA, 1999. MIT Press.

J. J. More and G. Toraldo. On the solution of large quadratic programming problems with bound counstraints. *SIAM Journal on Optimization*, 1(1):93–113, 1991.

Y. Nesterov and A. Nemirovskii. *Interior Point Algorithms in Convex Programming*. Number 13 in Studies in Applied Mathematics. SIAM, Philadelphia, 1993.

B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

A. J. Smola. *Learning with Kernels*. PhD thesis, Technische Universität Berlin, 1998. GMD Research Series No. 25.

R. J. Vanderbei. Robust optimization : Solution methologies with interior point methods. Talk held at the orsa/tims joint national meeting in orlando, fl, usa, Dept. of Civil Engineering and Operations Research, Princeton University, Princeton, NJ 08544, USA, April 1992.

R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic, Hingham, 1997.

R. J. Vanderbei, A. Duarte, and B. Yang. An algorithmic and numerical comparison of several interior-point methods. Technical Report SOR-94 - 05, Program in Statistics and Operations Research, Princeton University, 1994. URL ftp://columbia.princeton.edu/pub/rvdb/SOR-94-05.ps.Z.

S. V. N. Vishwanathan. *Kernel Methods: Fast Algorithms and Real Life Applications*. PhD thesis, Indian Institute of Science, Bangalore, India, November 2002. URL http://users.rsise.anu.edu.au/~vishy/papers/Vishwanathan02.pdf.

–

INDEX